

SAS/C[®]
Development
System
User's Guide





NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO.64 CARY, NC

POSTAGE WILL BE PAID BY ADDRESSEE

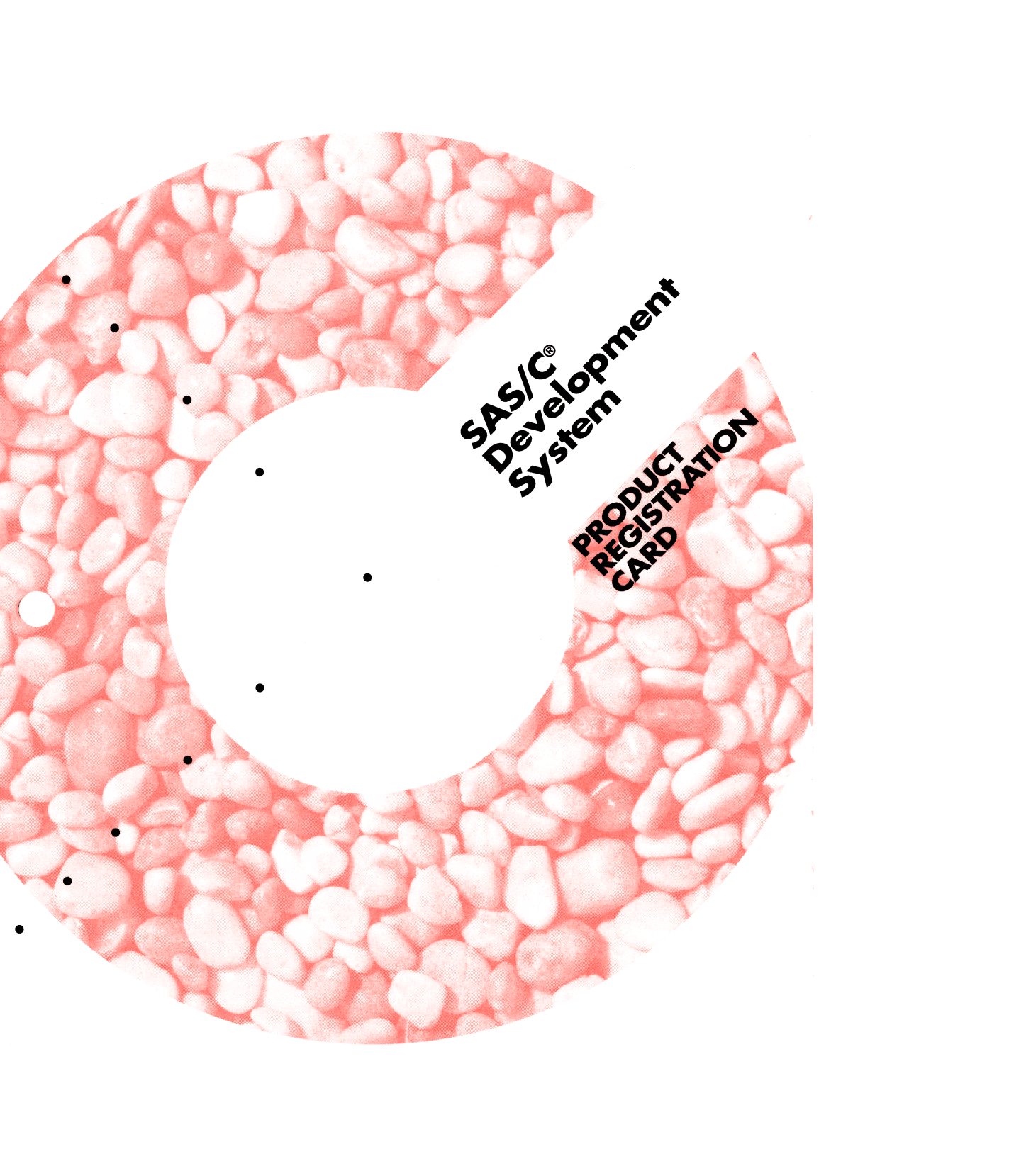
SAS INSTITUTE INC.
ATTN: BOOK SALES
SAS/C® DEVELOPMENT SYSTEM
100 SAS CAMPUS DRIVE
CARY, NC 27513
USA



A



B



**SAS/C[®]
Development
System**

**PRODUCT
REGISTRATION
CARD**

Become a SAS/C® Development System Registered User

Join the SAS Institute family as a registered user and receive the full benefits of service from SAS Institute:

- Access to the SAS Institute Technical Support Hotline.
- Direct notification of new versions and new products.

NOTE: If you are upgrading from a prior release of the SAS/C Development System and are already registered with SAS Institute, please do not return a new registration card. Your previous product serial number is still valid.

If you reside within the United States or outside of Europe, please complete and return registration card A. (If outside of the United States, please affix appropriate postage.)

If you reside in Europe, please complete and return registration card B. (Please affix appropriate postage.)

Protect your investment.

Complete the appropriate card and mail it today!

The number pre-stamped on this page is the same as the numbers stamped on the registration cards that are attached. This number is your product serial number. You will need this number when contacting Technical Support or Customer Service.

Serial Number: SAS6- 011 

SAS/C® Development System

The attached cards are your SAS/C Development System Registration Cards. You should only return one of them. It is for your protection and is informational only. This is not a warranty registration. If you have more than one product, please register each product using a separate card.

Product Information

SAS/C Development System Version 6._____

Date Purchased Purchased From
____ / ____ / ____ _____

A

Customer Information

Name _____

Company _____

Shipping Address _____

City _____ State _____ ZIP _____

Country _____ Telephone _____

Serial Number:

SAS6- 011 XXXXXXXXXX

B

**SAS/C[®] Development System
User's Guide, Volume 1:**
Introduction, Compiler, Editor,
Version 6.0

SAS/C[®] Development System User's Guide, Volume 1:
Introduction, Compiler, Editor, Version 6.0

■ **SAS/C[®] Development System User's Guide**

Volume 1:

Introduction, Editor, Compiler

Version 6.0



SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/C® Development System User's Guide, Volume 1: Introduction, Editor, Compiler, Version 6.0*, Cary, NC: SAS Institute Inc., 1992. 316 pp.

SAS/C® Development System User's Guide, Volume 1: Introduction, Editor, Compiler, Version 6.0

Copyright © 1992 by SAS Institute Inc., Cary, NC, USA.
ISBN 1-55544-496-2

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.
1st printing, July 1992

The SAS® System is an integrated system of software providing complete control over data access, management, analysis, and presentation. Base SAS software is the foundation of the SAS System. Products within the SAS System include SAS/ACCESS®, SAS/AF®, SAS/ASSIST®, SAS/CPE®, SAS/DMI®, SAS/ETS®, SAS/FSP®, SAS/GRAPH®, SAS/IML®, SAS/IMS-DL/I®, SAS/OR®, SAS/QC®, SAS/REPLAY-CICS®, SAS/SHARE®, SAS/STAT®, SAS/TOOLKIT®, SAS/CALC®, SAS/CONNECT®, SAS/DB2®, SAS/EIS®, SAS/ENGLISH®, SAS/INSIGHT®, SAS/LAB®, SAS/LOOKUP®, SAS/NVISION®, SAS/PH-Clinical®, SAS/SQL-DS®, and SAS/TUTOR® software. Other SAS Institute products are SYSTEM 2000® Data Management Software, with basic SYSTEM 2000, CREATE®, Multi-User®, QueX®, Screen Writer®, and CICS interface software; NeoVisuals® software; JMP®, JMP IN®, and JMP Serve® software; SAS/RTERM® software; and the SAS/C® Compiler and the SAS/CX® Compiler. MultiVendor Architecture™ and MVA™ are trademarks of SAS Institute Inc. SAS Institute also offers SAS Consulting® and On-Site Ambassador™ services. SAS Communications®, SAS Training®, SAS Views®, the SASware Ballot®, and Observations™ are published by SAS Institute Inc. All trademarks above are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. The Institute is a private company devoted to the support and further development of its software and related services.

Amiga Includes and Development Tools Version 2.0

Copyright © 1990 Commodore-Amiga, Inc. All rights reserved.

Amiga Workbench™ Version 2.0

Copyright © 1988, 1990 Commodore-Amiga, Inc. All rights reserved.

Installer

Copyright © 1991-1992 Commodore-Amiga, Inc. All rights reserved.

AmigaGuide™ and WDisplay

Copyright © 1991-1992 Commodore-Amiga, Inc. All rights reserved.

Distributed under license from Commodore.

Enforcer and Mungwall

Copyright © 1990-1992 Commodore-Amiga, Inc.

Distributed with permission from Commodore.

Amiga®, AmigaDOS®, Workbench®, and AmigaGuide™ are registered trademarks or trademarks of Commodore-Amiga, Inc. Commodore® is a registered trademark or trademark of Commodore Electronics Limited.

IBM® is a registered trademark of International Business Machines Corporation.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Contents

ix Reference Aids

xi Credits

xiii Acknowledgments

xv Using This Book

xxiii Changes and Enhancements

Part 1 Getting Started

Page 3 Chapter 1 Installing Your SAS/C Development System

3 Introduction

3 Protecting Your Investment

4 Installing the Product on A Hard Disk

8 Installing the Product on Floppy Disks

10 Assigning Logical Device Names

10 Setting Environment Variables

13 Chapter 2 Using Your SAS/C Development System

13 Introduction

14 Using SAS/C Tools from the Workbench Screen

22 Using SAS/C Tools from the Shell

25 Using Icons

27 Chapter 3 Getting Help

27 Introduction

27 Using the Help System

29 Contacting the Technical Support Division

Part 2 Using the Screen Editor (se)

37 Chapter 4 Using se

37 Introduction

38 Starting the Editor

39 Reading the Status Line

40 Moving Around in a File

40 Entering and Editing Text

46 Saving Your File and Exiting se

47 Managing Windows

48 Compiling from within the Editor

48 Getting Help

49 Chapter 5 Controlling se

49 Introduction

50 Selecting Menu Options

51 Moving Around in the File

52 Inserting Text

52 Deleting Text

53 Working with Blocks of Text

53 Searching for and Replacing Strings

54 Managing Windows

54 Changing Colors

55 Creating and Using Keystroke Macros

55 Editing a New File

56 Naming Files

56 Undoing Changes

56 Saving Your File and Exiting the Editor

57 Chapter 6 Customizing the Editor

57 Introduction

57 Displaying the Configuration Window

59 Displaying Key Definitions

60 Setting Key Definitions

61 Setting Tab Stops

61 Setting Other Options

64 Saving the New Settings

64 Reusing Saved Settings

65 Chapter 7 Using the AREXX Interface to se

65 Introduction

65 Executing A Series of Editor Commands

68 Communicating with External Programs

69 AREXX Macros Provided with the Compiler

70 AREXX Command Summary

Part 3 Using the Compiler and Linker

77 Chapter 8 Compiling and Linking Your Program

77 Introduction

77 Compiling Your Program

120 Linking Your Program

127 Using Overlays

135	Chapter 9 Running Your Program from the Workbench Screen
135	Introduction
135	Working with argc and argv
137	Managing the Standard I/O Window
139	Chapter 10 Using Startup Modules, Autoinitialization, and Autotermination Functions
139	Introduction
140	Understanding Startup Modules
140	Modifying <code>__main</code>
141	Linking with a Startup Module
145	Writing Applications without a Startup Module
145	Using Autoinitialization and Autotermination Functions
147	Initializing System Library Bases
153	Chapter 11 Using Amiga Specific Features of the SAS/C Language
153	Introduction
154	Using Special Keywords
160	Managing Stack Space
161	Using <code>#pragma</code> Statements
163	Using Unnamed Unions
164	Using Implicit Structure References
164	Using Equivalent Structures
166	Using Zero-Length Arrays
167	Specifying the Size of Enumerated Types
168	Using the <code>sizeof</code> and Comma Operators in Preprocessor Directives
168	Using C++ Style and Nested Comments
168	Using National Characters in Variable Names
168	Using Common Model External Data
171	Chapter 12 How Does the Compiler Work?
171	Introduction
171	Compiling Your Program
173	Linking Your Program
174	Running Your Program
175	Changing Hunk Names
176	Addressing Data
177	Understanding Data Types and Sizes
179	Storing Data

181 Chapter 13 Writing Portable Code

- 181 Introduction
- 181 Compiling Your Program
- 182 Writing Your Program

Part 4 Appendices

193 Appendix 1 Solving Common Problems

- 193 Introduction
- 193 Resolving Undefined Symbols
- 195 Fixing Compilation Errors
- 197 Using CodeProbe
- 197 Using Formatted Print Functions
- 198 Using getchar
- 199 Getting Incorrect Results from Function Calls
- 200 Crashing the Machine
- 200 Using The asctime Function
- 201 Managing the Standard I/O Window
- 201 Linking Resident Programs or Creating Resident Libraries
- 202 Writing Replacement Functions for SAS/C Library Functions
- 202 Eliminating Informational Messages

203 Appendix 2 Error and Warning Messages

- 203 Introduction
- 203 Using Error and Warning Messages
- 204 Explanations of Unnumbered Compiler Messages
- 206 Explanations of Numbered Compiler Messages
- 258 Explanations of Linker Messages
- 265 Enabling Suppressed Messages

267 Appendix 3 Implementation-Defined Behavior

- 267 Introduction
- 268 F.3.1 Translation
- 268 F.3.2 Environment
- 268 F.3.3 Identifiers
- 269 F.3.4 Characters
- 269 F.3.5 Integers
- 270 F.3.6 Floating-Point
- 270 F.3.7 Arrays and Pointers
- 271 F.3.8 Registers
- 271 F.3.9 Structures, Unions, Enumerations, and Bit-Fields
- 272 F.3.10 Qualifiers
- 272 F.3.11 Declarators
- 272 F.3.12 Statements
- 272 F.3.13 Preprocessing Directives

273	F.3.14 Library Functions
278	F.3.15 Locale-Specific Behavior
281	Appendix 4 Converting From Aztec C Options to SAS/C Options
285	Appendix 5 Converting from Version 5 to Version 6
285	Introduction
285	Upgrading from Previous Versions
286	Converting Compiler Options
294	Specifying Version 6 Libraries
295	Index

Reference Aids

Displays

- 17 2.1 The SAS/C Compiler Options Index Screen
- 18 2.2 The SAS/C Message Browser Window
- 19 2.3 The Edit Window Showing example.c
- 21 2.4 CodeProbe Windows with example.c
- 28 3.1 Editor Help Screen
- 39 4.1 Editing hello.c and example.c
- 58 6.1 se Configuration Window
- 59 6.2 se Configuration Window Showing Ctrl-F4

Figures

- 128 8.1 An Example Overlay Tree
- 130 8.2 Example Overlay Tree with Filenames
- 132 8.3 Overlay Tree for hw
- 147 10.1 Startup Modules, Autoinitialization Functions, and
Autotermination Functions

Tables

- 70 7.1 AREXX Commands with Keystroke Equivalents
- 73 7.2 AREXX Commands without Keystroke Equivalents
- 80 8.1 Summary of Compiler Options
- 119 8.2 Preprocessor Symbols Defined by the Compiler
- 120 8.3 Preprocessor Symbols Defined by Compiler Options
- 148 10.1 Library Bases for .library Files
- 281 A4.1 Converting From Aztec C Options to SAS/C Options
- 287 A5.1 Converting Version 5 Options to Version 6 Options
- 294 A5.2 Specifying Version 6 Libraries

Credits

Software

The SAS/C Development System is designed, built, and tested through the combined effort of many people at SAS Institute. The AmigaDOS development group includes the following individuals:

Development	Steve Krueger, Douglas Walker, Michael S. Whitcher
Testing and Technical Support	Twilah K. Blevins, Jim Cooper, Bob Patten, Khristi Tomlinson

These individuals continue to support future development of the SAS/C Development System.

Documentation

Composition	Candy R. Farrell, Denise T. Jones, Blanche W. Phillips, Pamela A. Troutman
Graphic Design	Creative Services Department
Proofreading	Heather B. Dees, Laurie J. Medley, Karen A. Olander, Josephine P. Pope, Christian Schwoerke
Technical Review	Jim Cooper, Steve Krueger, Khristi Tomlinson, Douglas Walker, Michael S. Whitcher
Writing and Editing	Jeanne Ferneyhough, Steve Krueger, Jeffrey Lopes, Khristi Tomlinson, Douglas Walker, Helen Weeks, John M. West

Acknowledgments

Many people make significant and continuing contributions to the development of the SAS/C Development System. First among them is John A. Toebe VIII, who was instrumental in the development of Versions 4.0 and 5.0 and who directed the early development of Version 6.0.

Others who have contributed to the development and testing of Version 6.0 of the SAS/C Development System include Aaron Avery, Ralph Babel, Jim Biggs, Bruce M. Drake, Andy Finkel, Nathan S. Fish, Jose M. Gallego, John Gerlach Jr., Maximilian Hantsch, Randell Jesup, David N. Junod, Andrew Kopp, Martin J. Laubach, John Lindwall, Christian Ludwig, Andrew N. Mercier, Mike Meyer, Tony Preston, Julius Oklamcak, Joe Porkka, Loren J. Rittle, Tomas Rokicki, Larry Rosenman, Carolyn Scheppner, Steve Shireman, Michael Sinz, Martin Taillefer, Steve Tibbett, John Wiederhirn, Loren Wilton, and Kenneth Yarnall.

The final responsibility for the SAS/C Development System lies with SAS Institute alone. We hope that you will always let us know your feelings about our product and its documentation. It is through such communication that the progress of SAS software continues to be accomplished.

Using This Book

Purpose

SAS/C Development System User's Guide, Volume 1: Introduction, Editor, Compiler, Version 6.0 describes how to install and use your SAS/C Development System. This book describes the editor and compiler in detail and provides an overview of some of the major tools available to the SAS/C programmer under AmigaDOS, such as the debugger, the message browser, and the `scopts` utility.

"Using This Book" describes how you can best use this book. It describes the book's intended audience, the audience's prerequisite knowledge, the book's organization and its conventions, and additional documentation that is available to you.

Audience

This book assumes you have a working knowledge of the C language and the AmigaDOS operating system. For a list of publications you can use to learn the C language or the AmigaDOS operating system, refer to the "Additional Documentation" section at the end of "Using This Book."

How to Use This Book

This section gives an overview of the book's organization and content. The book's chapters are described, followed by a section on how to use each chapter.

Organization *SAS/C Development System User's Guide, Volume 1: Introduction, Editor, Compiler* is divided into four parts; the chapters and appendices contained in each part are described here:

Part 1: Getting Started

Part 1 contains three chapters that show you how to install the SAS/C Development System, use its major features, and get help when necessary.

Chapter 1, "Installing Your SAS/C Development System" describes how to install the SAS/C Development System, assign logical device names, set environment variables, and upgrade from previous versions of the product.

Chapter 2, “Using Your SAS/C Development System”

describes briefly how to use the major features of the SAS/C Development System: the editor, the compiler, the debugger, the `scmsg` utility, and the `scopts` utility.

Chapter 3, “Getting Help”

describes how to use the online help system and how to contact the Technical Support Division.

Part 2: Using the Screen Editor (`se`)

Part 2 contains four chapters that describe how to use the editor, how to customize the keymap, and how to use AREXX with the editor.

Chapter 4, “Using `se`”

describes the most frequently used `se` commands and options.

Chapter 5, “Controlling `se`”

lists all of the options and commands available in `se`.

Chapter 6, “Customizing the Editor”

describes how to use the `se` configuration window. The configuration window enables you to customize the meaning of various keystrokes within `se`.

Chapter 7, “Using the AREXX Interface to `se`”

describes how to use AREXX to create complicated macros and to communicate with other programs from `se`.

Part 3: Using the Compiler and Linker

Part 3 contains six chapters that describe how to compile, link, and run your program. These chapters also provide guidelines for writing your application.

Chapter 8, “Compiling and Linking Your Program”

describes how to use the compiler and linker.

Chapter 9, “Running Your Program from the Workbench Screen”

describes special considerations for running your program from the Workbench screen.

Chapter 10, “Using Startup Modules, Autoinitialization, and Autotermination Functions”

describes what functions the startup modules perform and how to write your own autoinitialization and autotermination functions.

Chapter 11, “Using Amiga Specific Features of the SAS/C Language”

describes how to use SAS/C extensions to the C Language.

Chapter 12, “How Does the Compiler Work?”

describes briefly the format of the object files produced by the compiler. This chapter also describes what happens when you compile, link, and run your program.

Chapter 13, “Writing Portable Code”

provides guidelines for improving the portability of your program.

Part 4: Appendices

Part 4 contains five appendices.

Appendix 1, “Solving Common Problems”

contains answers to the questions that users ask most frequently when they call the Technical Support Division.

Appendix 2, “Error and Warning Messages”

contains explanations for the diagnostic messages produced by the compiler and linker.

Appendix 3, “Implementation-Defined Behavior”

describes how the SAS/C Compiler behaves for each of those areas where the ANSI Standard allows an implementation to define its own behavior.

Appendix 4, “Converting from Aztec C Options to SAS/C Options”

lists each of the Version 5 Aztec C options and their equivalent Version 6 SAS/C options.

Appendix 5, “Converting from Version 5 to Version 6”

lists each of the Version 5.10 options and their equivalent Version 6 options. This appendix also provides guidelines for converting your projects to Version 6 SAS/C software.

What You Should Read The following table points you to specific chapters in this book for information on particular topics. For other references, refer to “Additional Documentation” later in this section.

For information on	You should read
installing the product	Chapter 1
upgrading from previous versions of the product	Chapter 1 and Appendix 5
using the major features of the SAS/C Development System	Chapter 2
contacting the Technical Support Division	Chapter 3
using online help	Chapter 3
using the editor	Chapters 4 and 5
specifying the <code>sc</code> command and compiler options	Chapter 8
managing the environment in which your program runs	Chapters 9 and 10
using extensions to ANSI C	Chapter 11

Conventions

This section covers the typographical and syntax conventions used in this book.

Typographical Conventions The SAS/C books for AmigaDOS use special fonts to depict specific types of information. These typographical conventions are as follows:

- `roman` is the basic type style used for most text.
- `monospace` is used to show example statements or programs. Monospace is used also for items specific to the C language, such as the names of functions, header files, and keywords.
- oblique* is used for arguments or variables whose values are supplied by the user; for example, you should enter an appropriate filename when you see *filename*.
- italic* is used for terms that are defined. Italic is also used to indicate arguments for which you supply a value.

**Syntax
Conventions**

This book uses the following conventions for syntax:

monospace	indicates commands, keywords, and switches that should be spelled exactly as shown. These arguments may or may not be optional, depending on whether they are enclosed in square brackets.
<i>italic</i>	indicates arguments for which you supply a value.
[] (square brackets)	indicate an optional argument when they surround the argument.
. . . (ellipsis)	indicates that you can repeat the argument or group of arguments preceding the ellipsis any number of times.
 (vertical bar)	means to choose one item from a group of items separated by the bars.

The following example illustrates these syntax conventions:

env [*function* | *integer*]

env

is a command name, so it appears in monospace type.

function

is a function for which you supply the name, so it appears in italic type.

[*function* | *integer*]

are both optional, so they are enclosed in square brackets.

function | *integer*

indicates that you can specify only one of the items separated by the vertical bar.

Additional Documentation

The following sections list documentation you may find helpful in using the SAS/C Development System.

**SAS
Documentation**

If you are interested in SAS documentation, you need to contact the Book Sales department by writing to the following address or by calling the following number:

SAS Institute Inc
Book Sales Department
SAS Campus Drive
Cary, NC 27513
919-677-8000

SAS/C Development System

This book is part of a set of publications for the SAS/C Development System. There are three other publications in the set:

- *SAS/C Development System User's Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0* describes how to use
 - CodeProbe, the SAS/C Source Level Debugger
 - the utilities such as **smake**, **grep**, **scopts**, and **scmsg**
 - the SAS/C Macro Assembler
 - the global and peephole optimizers.
- *SAS/C Development System Library Reference, Version 6.0* describes how to
 - access libraries
 - create your own libraries
 - use header files to reduce the amount of time and space required by your program.
- *SAS/C Development System Quick Reference, Version 6.0* contains reference tables for the library functions, compiler and linker options, and debugger commands.

These volumes are sold together as a set with the software. You can order the entire package by specifying order #A56185.

Other Documentation

The following sections list additional reference material specific to the C language, Amiga and AmigaDOS programming, and the Motorola 68xxx microprocessor family.

C Language

The following books describe the C programming language:

- American National Standards Committee (1990), *American National Standard for Information Systems—Programming Language C*, Document Number X3J11/90-013, Washington, D.C.: X3 Secretariat: Computer and Business Equipment Manufacturers Association.
- Harbison, Samuel P. and Steele, Guy L., Jr. (1990), *C: A Reference Manual, Third Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.
- Kernighan, Brian W. and Ritchie, Dennis M. (1988), *The C Programming Language, Second Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.
- Kochan, Stephen G. (1988), *Programming in ANSI C, First Edition*, Indianapolis, Indiana: Hayden Books.

Amiga and AmigaDOS

The following books provide information specifically about programming on the Amiga system. Most of the examples in these books are written in C or assembly language.

Commodore-Amiga, Inc. (1990), *Using the System Software*, Reading, MA: Addison-Wesley Publishing Company.

Commodore-Amiga, Inc. (1991), *Amiga Hardware Reference Manual, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.

Commodore-Amiga, Inc. (1991), *Amiga ROM Kernel Reference Manual: Devices, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.

Commodore-Amiga, Inc. (1991), *Amiga ROM Kernel Reference Manual: Includes and Autodocs, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.

Commodore-Amiga, Inc. (1991), *Amiga User Interface Style Guide*, Reading, MA: Addison-Wesley Publishing Company.

Commodore-Amiga, Inc. (1991), *The AmigaDOS Manual, 3rd Edition*, New York, NY: Bantam Books.

Commodore-Amiga, Inc. (1992), *Amiga ROM Kernel Reference Manual: Libraries, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.

Motorola 68xxx The following books contain information specifically about programming the Motorola 68xxx family of microprocessors.

Motorola, Inc. (1987), *MC68881/MC68882 Floating-Point Coprocessor User's Manual, First Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Motorola, Inc. (1989), *MC68030 Enhanced 32-Bit Microprocessor User's Manual, Second Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Motorola, Inc. (1989), *Programmer's Reference Manual*, Phoenix, AZ: Motorola Literature Distribution.

Contacting CATS You can contact Commodore Application and Technical Support at the following address:

1200 Wilson Drive
West Chester, PA 19380
215-429-0643

Changes and Enhancements

The following sections describe some of the changes and enhancements for the SAS/C Development System, Version 6.0.

- The SAS/C Development System has an extensive online help system implemented using the AmigaGuide hypertext system from Commodore. The online help describes each utility, CodeProbe command, editor command, library function, diagnostic message, and compiler option. For a complete description of using the help system, see Chapter 3, “Getting Help.”
- The SAS Institute Technical Support Division has a new facility called EMITS (Electronic Mail Interface to Technical Support) that allows you to report problems and receive help through Internet. For more information, see Chapter 3, “Getting Help.”
- Compiler executables and other files are installed in a directory tree pointed to by the assign `sc:`. Unlike the Version 5 assign `lc:`, `sc:` points to the root of the compiler’s installation directory.
- The `lc` compiler front-end has been replaced with the new `sc` front-end, which takes options in a different form. The new `sc` front-end accepts C source files, assembly language files, object files, and libraries. You may specify options before or after the filenames. See Chapter 8, “Compiling and Linking Your Program,” for more information.

To help you make the transition to Version 6, the SAS/C Development System provides two utilities, `sc5` and `lctosc`. The `sc5` command accepts options in the form accepted by `lc` and invokes the Version 6 compiler. The `lctosc` utility accepts options in the form accepted by `lc` and prints the equivalent `sc` options to `stdout`. Both `sc5` and `lctosc` read the `sasopts` file, if present. Appendix 5, “Converting from Version 5 to Version 6,” provides guidelines for converting your projects to Version 6.

- The names of various other executables have changed. The following table lists the names of the Version 5 executables and the corresponding Version 6 names.

Version 5	Version 6
blink	slink
lse	se
sascpts	scpts
sascsetup	scsetup

- Many symbols and functions in the link libraries have new names that adhere to the ANSI Standard. The Standard requires that all non-ANSI symbols referenced in ANSI header files start with two underscores or an underscore and a capital letter. The Version 5 symbols that violated this standard have been renamed, usually by adding one or two underscores to the front of the name. For information on a specific symbol, refer to Chapter 6, “Predefined Data Name Reference,” in *SAS/C Development System Library Reference*.
- The name of the file that is created by the **scpts** utility has been changed from **sascpts** to **scptions**.
- The AmigaDOS 1.3 header files are no longer shipped with the SAS/C Compiler. However, all AmigaDOS 1.3 functions are available under AmigaDOS 2.0, so you can develop code that runs under AmigaDOS 1.3 by restricting your program from using any AmigaDOS 2.0 functions.
- The SAS/C Development System, Version 6, does not have separate libraries for use with registerized parameters. The normal link libraries have both registerized and nonregisterized entry points.

If you write a function that has the same name as a library function, you need to add the **__regargs** keyword to your function or compile with the **parms=both** or **parms=register** option. For example, you may want to replace the SAS/C library function **malloc** with your own version of **malloc**. If you compile with the **parms=stack** option or define your version of **malloc** with the **__stdargs** keyword, then two versions of **malloc** are linked into your executable. If you use other SAS/C library functions that call **malloc**, these functions use the version of **malloc** in the SAS/C libraries. However, your functions that call **malloc** use your version of **malloc**. To make sure that all calls to **malloc** are using your version

of **malloc**, define your version with **__regargs** or compile with the **parms=both** or **parms=register** option.

For information on using registerized parameters, refer to the description of the **__regargs** keyword in Chapter 11, “Using Amiga Specific Features of the SAS/C Language,” and the description of the **parameters** compiler option in Chapter 8, “Compiling and Linking Your Program.”

- The **lc1**, **lc2**, and **go** commands are not included in Version 6 of the SAS/C Development System. The **sc** driver loads AmigaDOS shared libraries to handle the work of the various phases instead of invoking multiple programs.
- The **lseinst** program has been removed, and the **se** configuration window added in its place. Select the Configuration Window option from the Options pull-down menu in **se** to invoke the configuration window. For complete description of using the configuration window, see Chapter 6, “Customizing the Editor.”
- The precompiled header files in Version 5 have been replaced with GSTs (Global Symbol Tables). GSTs are much faster than precompiled headers because they remain in RAM between compiles. Some additional utilities, **gst** and **hypergst**, allow you to browse the RAM-resident GST files for information on symbols defined in system header files or in your header files. For information on creating and using GSTs refer to *SAS/C Development System Library Reference* and to the descriptions of the **gst** and **hypergst** utilities in *SAS/C Development System User's Guide, Volume 2: Debugger, Utilities, Assembler*. See also the descriptions of the **gst** and **makegst** compiler options in Chapter 8, “Compiling and Linking Your Program.”
- The new **scmsg** utility enables you to integrate any editor that supports AREXX into the SAS/C Development System. For more information, see the description of the **errorrexx** compiler option in Chapter 8, “Compiling and Linking Your Program,” and the description of the **scmsg** utility in *SAS/C Development System User's Guide, Volume 2*.
- The global optimizer supports many new optimizations, including inline functions. You can use the **__inline** keyword to specify a function that is to be inlined. See Chapter 11, “Using Amiga Specific Features of the SAS/C Language,” for information on special keywords supported by the SAS/C Compiler. Refer to *SAS/C Development System User's Guide, Volume 2* for information on the global optimizer.

- The new peephole optimizer improves code quality significantly. The peephole optimizer is run automatically when you choose the **optimize** compiler option, unless you specifically suppress it with **nooptpeep**. Refer to *SAS/C Development System User's Guide, Volume 2* for information on the peephole optimizer.
- The CodeProbe debugger has numerous enhancements. The major enhancements include the following:
 - The command syntax has been totally rewritten and greatly extended.
 - The debugger has new support for debugging AmigaDOS shared libraries.
 - The debugger automatically detects Enforcer hits caused by your program and halts your program at the location of the hit.
 - The debugger now has cross-debugging capability. This new feature allows you to run your program on one machine and debug the program using another machine connected through the serial port or a network file system.

Refer to *SAS/C Development System User's Guide, Volume 2* for a complete description of using the debugger.

- The compiler still generates code by default to check for stack overflow at the entry to each function. However, this code is more reliable under Version 6.0.

The compiler also supports a new option that generates code to allocate a new stack and allow your program to continue running if the old stack runs out. For more information, see the description of the **stackextend** compiler option in Chapter 8, "Compiling and Linking Your Program." See also the description of the `__stackextend` keyword in "Managing Stack Space" in Chapter 11, "Using Amiga Specific Features of the SAS/C Language."

- The compiler now supports the common model for external data as well as the strict reference-definition model. For more information, see Chapter 11, "Using Amiga Specific Features of the SAS/C Language."
- The compiler now supports a **coverage** option that generates special code that tells you which portions of your program were executed by your test cases. A new utility, the **cover** utility, analyzes the data produced by the compiler when you specify **coverage**. This information helps you design test cases that test all portions of your code. For more information, see the description of the **coverage** compiler option in Chapter 8, "Compiling and Linking Your Program," and the description of the **cover** utility in *SAS/C Development System User's Guide, Volume 2*.

- The **stringmerge** compiler option, equivalent to the **-cs** in Version 5, merges all string constants and places them in the code section. Unlike the Version 5 option, **stringmerge** also places all data declared **static const** and all automatic initializer data in the code section. For more information, see the description of the **stringmerge** compiler option in Chapter 8, “Compiling and Linking Your Program.”
- The compiler can now produce a disassembly of the generated object code. For more information, see the description of the **disassem** compiler option in Chapter 8, “Compiling and Linking Your Program.”
- The compiler now adjusts dynamically to low-memory situations. If you compile your program and your machine runs low on memory, the compiler displays the message *****Freeing Resources** and frees memory to enable it to continue the compilation. You can also use the **memorysize** compiler option to limit the compiler’s memory use. For more information, see the description of the **memorysize** option in Chapter 8, “Compiling and Linking Your Program.”
- You can designate autoinitialization functions that you want the compiler to call automatically before it calls your **main** routine. You can also designate autotermination functions that you want the compiler to call after **main** returns. For more information, see Chapter 10, “Using Startup Modules, Autoinitialization, and Autotermination Functions.”
- System library bases that are not defined in your code are automatically opened and initialized. For more information, see Chapter 10, “Using Startup Modules, Autoinitialization, and Autotermination Functions.”
- The SAS/C Compiler now permits references to unnamed unions and direct references to members of substructures. The compiler also permits zero-length arrays as members of structures. For more information, see Chapter 11, “Using Amiga Specific Features of the SAS/C Language.”
- The new **#pragma tagcall** allows you to call AmigaDOS functions that take a variable parameter list without using assembler stubs. For more information, refer to *SAS/C Development System Library Reference*.

- The new **#pragma msg** allows you to control compiler diagnostic output more closely. For more information, see Chapter 11, “Using Amiga Specific Features of the SAS/C Language.”
- The compiler now supports **char**, **short**, and **long enum** types. For more information, see Chapter 11, “Using Amiga Specific Features of the SAS/C Language.”

■ Part 1

Getting Started

- Chapters**
- 1 Installing Your SAS/C® Development System
 - 2 Using Your SAS/C® Development System
 - 3 Getting Help

1 Installing Your SAS/C® Development System

- 3 *Introduction*
- 3 *Protecting Your Investment*
- 4 *Installing the Product on A Hard Disk*
 - 5 *Modifying Your Startup File*
 - 6 *What Directories Does the Installation Program Create?*
- 8 *Installing the Product on Floppy Disks*
- 10 *Assigning Logical Device Names*
- 10 *Setting Environment Variables*

Introduction

This chapter describes how to install your SAS/C Development System and provides guidelines for upgrading your projects to Version 6.

Protecting Your Investment

Before you install your SAS/C Development System, you should do the following:

- Make backup copies of the disks distributed with this product. To make copies, you can use the AmigaDOS **diskcopy** command, or you can select the disk icon and choose **copy** from the Icons menu. Refer to your AmigaDOS manual for more information on copying disks.
- Write down your registration number where you can find it easily. You must give your registration number to the Technical Support staff before you can receive help from the SAS Institute Technical Support Division.
- Complete the registration card that comes with this product and return it to SAS Institute Inc. You must register your purchase to qualify for technical support and to be notified about future upgrades.

If you are upgrading from a previous version, and you have already registered the previous version, you do not need to send in a new registration card. If you are not sure if you are in the SAS Institute database of registered users, write your old registration number and a brief explanation on the new card before sending in the card.

Installing the Product on A Hard Disk

The installation program is distributed on disk number 1. You should run the installation program from the Workbench screen, not the Shell.

To install the SAS/C Development System, insert disk number 1 in the disk drive and double-click on the disk icon to open the `SASC_6.0_Disk_1` window. Double-click on the `Install-SASC` icon to open the Install-SASC window.

If you want to see exactly what the installation program will do before you actually install the product, you can select the Pretend to Install and Log File options, and then proceed through the installation process. As you go through the installation process, the installation program writes to a log file all of the commands it would have executed if you were actually installing the product. The log file is named `install_log_file` and is written to the directory in which you are installing the product (or to `ram:`, if it cannot write to that directory).

As you go through the installation process, the installation program asks you some questions, including the name of the directory, or destination, where you want to install the SAS/C Development System. The installation program allows you to save disk space by asking you which parts of the product you want to install. For example, you may choose not to install the cross debugger. The installation program also allows you to choose whether you want to install compressed or normal C header files. Choose compressed header files only if you need to conserve space on your hard disk. If you choose not to install parts of the product, include this information in any communication you have with the Technical Support Division.

After you have answered all of the questions (and if you select Install for Real), the installation program then proceeds to install the product. The program prompts you to insert disks as necessary.

Context-sensitive online help is available at any point in the installation process by clicking on the `Help` button, and you can abort the installation process at any time by clicking on the `Abort` button.

After all the files on the distribution disks have been copied to your hard disk, the installation program asks you if you want the following statements added to your `s:user-startup` file.

```
assign sc: destination:sc
assign lib: sc:lib
assign include: sc:include
path sc:c add
```

The installation program adds these statements automatically if you select `Proceed`. If you decide to add these statements manually and do not

select **Proceed**, make sure you add them before any line that calls a utility that creates a new Shell, such as PopCLI.

Finally, the installation program displays the **read.me** file.

Note: Read the **read.me** file carefully. This file contains important information that is not contained in the documentation distributed with the product. This information may concern new or enhanced features, bug fixes, or changes to the documentation. The **read.me** file is installed with the rest of the product, so you can refer to this file later.

Modifying Your Startup File

If you are running AmigaDOS 2.0, the modifications that the installation program makes to your startup file (if you selected **Proceed**) are sufficient to allow the compiler to run.

If you are running AmigaDOS 1.3, you may need to make an additional change to your **s:startup-sequence** file to make sure that these **assign** and **path** statements take effect.

The installation program adds the **path** and **assign** statements to the file **s:user-startup**. The installation also asks you where to add the following lines:

```
if exists s:user-startup
    execute s:user-startup
endif
```

The installation program suggests a place to enter these lines, but you can specify a different place if necessary. Make sure that these lines are entered before any line that calls a utility that creates a new Shell, such as PopCLI. The installation program adds these lines to either **s:startup-sequence** or **s:startupII**.

If the lines are added to **s:startupII**, you must verify that **s:startupII** is properly executed from **s:startup-sequence**. Under some older versions of AmigaDOS Version 1.3, **s:startupII** is executed from **s:startup-sequence** through either a **newshell** command or a **run execute** command. Therefore, any **path** statements added to the **s:user-startup** file do not have any effect under the Workbench screen or the Shell. Edit your **s:startup-sequence** file, and verify that **s:startupII** is executed with an **execute** command only. If not, change the command to the following:

```
execute s:startupII
```

If this command is not in your **s:startup-sequence** file, add it. If **s:startupII** is executed with a **run execute** command, delete the word **run**.

After you have saved your `s:startup-sequence` file with the correct command and completed the installation process, re-boot your machine. Your compiler should work correctly.

What Directories Does the Installation Program Create?

After you finish installing the product, you will have a directory named `sc` that contains several files and subdirectories. The following list describes the directories that have icons.

Note: Some of these icons may not be present if you choose not to install the entire package.

- `starter_project` is the directory that the `scsetup` utility uses as a model whenever you set up a project. (A *project* consists of all of the files that make up an application.) You can add icons and programs to `starter_project` as necessary. For example, you can create an `scoptions` file in this directory that contains any compiler options that you want to be copied into each of your project directories. `scsetup` copies the entire contents of `starter_project` into the project drawer you specify. The installation program places the following icons (and files) in this drawer:

Edit invokes the `se` editor

Build builds the project

Debug runs the debugger

SCoptions sets compiler options.

- `c` contains the executable modules for the SAS/C Development System, including the assembler, compiler, debugger, linker, and various utilities.
- `icons` is a directory used by `scsetup`, `se`, and other utilities that contains default icons for different file types. You can replace any of the icons in this directory or add your own icons. For more information on using icons, see Chapter 2, “Using Your SAS/C Development System.”
- `examples` contains example programs. For each example, this directory contains complete source code and a `read.me` file that explains the example program and describes how to run the example. Some of the examples are in their own subdirectory under the `examples` directory.
- `help` contains help files used by the various SAS/C utilities. Chapter 3, “Getting Help,” describes the contents of this directory and how to use the help system.

The installation program also creates some directories that do not have icons. These directories are as follows:

- **env** contains default configuration information for tools such as the editor. The tools look first in **env:sc** for their configuration files, and if the files are not there, the tool looks in **sc:env**.
- **source** contains C and assembly language source code for various parts of the SAS/C Development System. For example, this directory includes the source code to the different startup modules.
- **include** contains C header files and assembler header files. The **INCLUDE:** logical device name should point to this directory.
- **Lib** contains link libraries. The **LIB:** assign should point to this directory.
- **libs** contains resident libraries used by the various SAS/C commands. These libraries must exist in the **sc:libs** drawer. Do not attempt to copy them into **libs:** with the AmigaDOS system libraries. The following libraries are included in this drawer:

sc1.library

is required by the compiler. This library must be present to compile any code.

sc2.library

is required by the compiler. This library must be present to compile any code.

scgo.library

is required by the global optimizer. This library must be present if you compile with the **optimize** option and do not specify the **nooptglobal** option.

scpeep.library

is required by the peephole optimizer. This library must be present if you compile with the **optimize** option and do not specify the **nooptpeep** option.

scdebug.library

generates debugging information. Without this library, the compiler can generate partial line number information but cannot generate any symbolic debugging information.

scgst.library

generates or accesses GSTs (Global Symbol Tables). This library is not required if you never use the GSTs. Refer to *SAS/C Development System Library Reference, Version 6.0* for information on using GSTs.

schi.library

is used by CodeProbe to read the debugging information in an executable module. This library is required if you run the debugger.

sekeymap.library

is used by the editor to allow you to modify the editor's configuration.

scspill.library

detects low-memory conditions and attempts to reduce the compiler's memory usage. This library is required if you want the compiler to react dynamically to low-memory conditions.

If you compile your program and your machine runs low on memory, the compiler displays the message *****Freeing Resources** and frees memory to enable it to continue the compilation. You can force the compiler to free memory at any time by pressing Control-F in the window to which the compiler is sending output.

sclist.library

generates listing and cross-reference files. This library is not required if you never use the **list** or **xref** compiler options.

It is recommended that you do not place your source code in the **sc** directory tree. Installing updates to the SAS/C Development System is easier if you can replace the entire directory without worrying about losing any of your own program files. Instead, you should set up a different directory for your C development work. You may find it useful to keep different projects in different directories. Keeping projects separated makes it easier to keep track of the **smakefiles** and compiler options required for each project. Remember, a *project* consists of all of the files that make up an application.

Installing the Product on Floppy Disks

The installation program is distributed on disk number 1. You should run the installation program from the Workbench screen, not the Shell.

Before installing the SAS/C Development System, make sure you have at least two blank disks on which to install this product.

To install the SAS/C Development System, insert disk number 1 in the drive **df0:** and double-click the disk icon to open the **SASC_6.0_Disk_1** window. Double-click the **Install-SASC** icon to open the Install-SASC window.

If you want to see exactly what the installation program will do before you actually install the product, you can select the Pretend to Install and

Log File options, and then proceed through the installation process. As you go through the installation process, the installation program writes to a log file all of the commands it would have executed if you were actually installing the product. The log file is named `install_log_file` and is written to `ram:`.

As you go through the installation process, the installation program asks you some questions, then proceeds to install the product. The program prompts you to insert disks as necessary.

The installation program produces the following two working disks:

- Disk 1 is a bootable disk that contains the compiler (`sc`), linker (`slink`), editor (`se`), and the `smake`, `scopts`, and `scmsg` utilities.
- Disk 2 contains compressed header files, CodeProbe, `sc.lib`, `scm.lib`, and a small version of `amiga.lib` containing only global symbols (no stubs).

The installation program asks you if you want to create a third disk:

- Disk 3 contains the remaining utilities and the global and peephole optimizers, if you choose to create disk 3. Under AmigaDOS 1.3, you will not be able to use the global and peephole optimizers. Under AmigaDOS 2.0, a multiple-directory `assign` statement is used to access the optimizers.

Online help is available at any point in the installation process, and you can abort the installation process at any time.

Finally, the installation program displays the `read.me` file.

Note: Read the `read.me` file carefully. This file contains important information that is not contained in the documentation distributed with the product. This information may concern new or enhanced features, bug fixes, or changes to the documentation.

Assigning Logical Device Names

When you install the SAS/C Development System (either on floppy disks or on a hard drive), the installation program tells you that specific **assign** statements must be added to your startup file. You may choose to allow the installation program to add these statements for you. The following list describes each of the logical device names that need to be added to your startup file.

sc: specifies the directory in which the compiler and other utility programs are located. For example, the following statement indicates that the compiler and utilities are in the directory **work:sc**.

```
assign sc: work:sc
```

include: specifies the directory in which the compiler can find system header files included in your programs by specifying the **#include** statement with angle brackets (**<>**). For example, you could enter the following **assign** statement from the Shell prompt:

```
assign include: df0:sc/include
```

Then, if you compile a program containing the statement **#include <stdio.h>**, the compiler attempts to include the file **df0:sc/include/stdio.h**.

lib: specifies the directory in which the linker can find the libraries and other modules needed to build executable files. For example, the following Shell command indicates that the libraries are in the directory **sc/lib** on the **work:** volume:

```
assign lib: work:sc/lib
```

Setting Environment Variables

The various tools in the SAS/C Development System sometimes require information in environment variables. For example, the **sc** command looks for default options in a file called **scoptions**; if the compiler does not find such a file in the current directory, it looks in the environment variable **ENV:sc/options**.

Environment variables are stored in the logical device **ENV:** in files with the same name as the variable. **ENV:** is usually assigned to the

ram: device or some other temporary location. You can set environment variables from the Shell with the **setenv** command.

All SAS/C utilities that use environment variables access them from the **ENV:sc** subdirectory, so be sure to create this subdirectory in **env:** each time you boot your machine. Under AmigaDOS 2.0, the installation program creates this subdirectory by creating the directory **ENVARC:SC**. Under AmigaDOS 1.3, **ENVARC:** does not necessarily exist, so you must make sure that the **ENV:SC** subdirectory is created each time you boot your machine.

2 Using Your SAS/C® Development System

- 13 *Introduction*
- 14 *Using SAS/C Tools from the Workbench Screen*
 - 14 *Setting Up a Project*
 - 15 *Creating a Source File with se*
 - 16 *Compiling, Linking, and Running Your Program*
 - 20 *Debugging Your Program*
- 22 *Using SAS/C Tools from the Shell*
 - 22 *Creating a Source File with se*
 - 22 *Compiling, Linking, and Running Your Program*
 - 25 *Debugging Your Program*
- 25 *Using Icons*

Introduction

This chapter introduces the primary features of the SAS/C Development System. It shows you how to perform the following tasks:

- ☐ set up a project
- ☐ set compiler options for the project
- ☐ create a C source file
- ☐ compile, link, and run a C program
- ☐ use the debugger to look at your program.

As you perform these tasks, the SAS/C Development System creates several icons. The icons it creates depend on the filename extension of the file you create. The last section in this chapter, “Using Icons,” describes the default icons for each of these files and how to customize these icons.

Each of the tools used to perform these tasks are described in more detail in subsequent chapters in this book and in *SAS/C Development System User's Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0*.

The SAS/C Development System provides an environment based on Intuition from which you can perform each of these tasks. However, you can also perform these tasks from the Shell.

Using SAS/C Tools from the Workbench Screen

The following sections describe how to use the SAS/C Development System from the Workbench screen. Specifically, these sections discuss the following:

- setting up a project using the **scsetup** utility
- creating a C source file using the **se** editor
- setting compiler options for the project using **scopts**
- compiling and linking a program using the Build icon
- running your program
- using the debugger to look at your program.

This chapter provides example directions that you can enter on your machine as you read through the chapter. These examples illustrate how the many features of the SAS/C Development System work together.

Setting Up a Project

To make your C development work easier, you can set up either a new or existing project using the **scsetup** utility. When you *set up* a project, **scsetup** creates icons for any files or directories that already exist in the project and copies the contents of the **sc:starter_project** directory into the project. The **starter_project** directory contains icons for each of the tools you need most often during your development work:

- SCOptions** starts the **scopts** utility and allows you to specify the compiler options with which you most often want to compile the programs in the directory you are setting up.
- Debug** starts the CodeProbe Source Level Debugger.
- Build** starts the **smake** utility. **smake** is a utility that helps you manage projects that are composed of many files.
- Edit** starts the editor, **se**. You can substitute your own editor for **se** by selecting Info from the Workbench pull-down menu and changing the default tool in the **Edit** icon from **sc:c/se** to your editor.

It is recommended that you do not place your source code in the **sc** directory tree. Installing updates to the SAS/C Development System is easier if you can replace the entire directory without worrying about losing any of your own program files.

To set up a project, run the **scsetup** utility by double-clicking the **scsetup** icon in the **sc:c** drawer. **scsetup** asks you for the name of the new project drawer (directory). For example, to create a new directory called **cstuff** on the **Work:** volume, you would enter:

```
Work:cstuff
```

scsetup creates a new directory called **cstuff** in **Work:** that contains the four icons described previously.

For more information about the **scsetup** utility, refer to *SAS/C Development System User's Guide, Volume 2*.

Creating a Source File with se

After you set up a directory for your C files, you can start entering your C programs. To create a new file, double-click on the **Edit** icon to start the **se** editor. **se** displays a window into which you can enter your program.

Type in the following program:

```
#include <stdio.h>
#include <math.h>
#include <proto/dos.h>

int main(void)
{
    int i=42;
    double d

    d=i/2;
    printf("i = %d, d=%2.1f\n",i,d);
    Delay(60);
    return(0);
}
```

Note: Enter this text exactly as shown. The semicolon missing from the declaration of **d** produces an error message that is used to illustrate **scmsg**, the message browser utility.

If you enter any text differently from that shown here, simply use the backspace or delete key to delete the incorrect characters and correct the error. When you have finished entering the program, you must name the file, save it, and close the editor window as follows:

1. Press and hold the right mouse button to display the **se** menu.
2. Point to the Project heading in the menu bar to open the Project menu.
3. Select the Rename option. At the bottom of the screen, **se** asks you to enter a new filename for the current file.
4. Type **example.c** and press Return.
5. Hold down the right Amiga key and press the letter S to save the file and close the editor window.

Under AmigaDOS 1.3, the icon for this new file is not displayed until you close and reopen the window for the directory in which you created the file. If necessary, close and reopen the window to display the icon for **example.c**.

For more information about the **se** editor, see Chapter 4, “Using se,” and Chapter 5, “Controlling se.”

Compiling, Linking, and Running Your Program

After you have created a C source program using the editor, you can compile, link, and run your program. If you want to compile and link your program with the default compiler options, simply double-click on the **Build** icon. However, if you want to change the default values or specify additional options, you can use the **scopts** utility. With **scopts**, you can set compiler options simply by clicking on the option.

To successfully run the example shown in this chapter, you need to set additional compiler options.

Setting Compiler Options with **scopts**

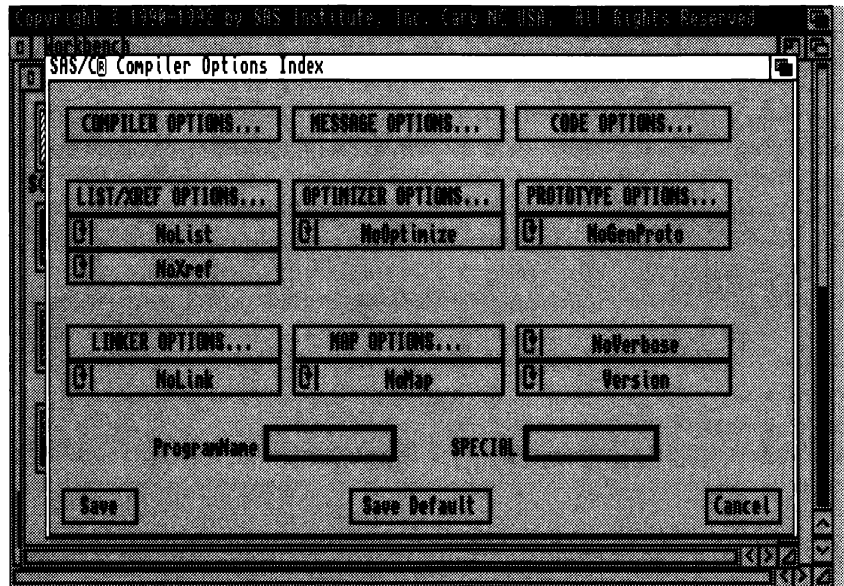
Usually, you compile all of the programs in a project with the same compiler options. The **scopts** utility allows you to click on the options with which you want to compile and to save the options in an **scoptions** file in the project drawer.

To run **scopts**, double-click on the **SCoptions** icon. **scopts** displays the first of several windows on which you can select the options with which you want to compile.

To run the **example.c** program, you must specify the **math=standard** and the **link** options. To use the message browser as described in the section “Compiling and Linking with the Build icon,” you must specify the **errorrexx** option. Also, to use the CodeProbe debugger as described in the section “Debugging Your Program,” later in this chapter, you must specify the **debug=symbolflush** option.

The **link** option appears on the first screen that **scopts** displays, the SAS/C Compiler Options Index screen, as shown in Display 2.1.

Display 2.1
The SAS/C Compiler
Options Index Screen



To select the **link** option, click once on the **NoLink** cycle gadget. The gadget changes colors, and the text now reads **Link**.

To select the **math=standard** option, click on the **CODE OPTIONS...** gadget, and then click once on the **NoMath** gadget. **NoMath** changes to **Math=STANDARD**. Click on OK to return to the SAS/C Compiler Options Index screen.

To select the **debug** option, click on the **COMPILER OPTIONS...** gadget, and then click three times on the **NoDebug** gadget. **NoDebug** changes to **Debug=SymbolFlush**. Click on the OK gadget to return to the SAS/C Compiler Options Index screen.

To select the **errorrexx** option, click once on the **MESSAGE OPTIONS...** gadget, and then click on the **NoErrorRexx** gadget. **NoErrorRexx** changes to **ErrorRexx**. Click on OK to return to the SAS/C Compiler Options Index screen.

To save the option settings, click on the **Save** gadget. The **scopts** utility saves the options you select in the file **scoptions** in the project directory. When you compile your program, the compiler reads the options from **scoptions**.

For more information about the `scopts` utility, refer to *SAS/C Development System User's Guide, Volume 2*.

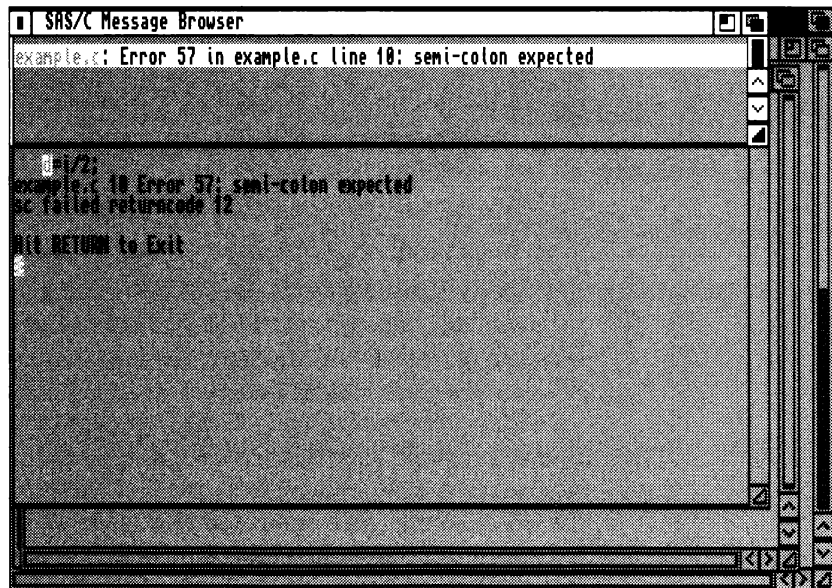
Compiling and Linking with the Build icon

After you have selected the appropriate options, you can compile and link your program by double-clicking the **Build** icon to start the `smake` utility.

`smake` examines the drawer and determines that it contains only one C source file, `example.c`. `smake` then invokes the `sc` command to compile `example.c` using the options you specified with the `scopts` utility.

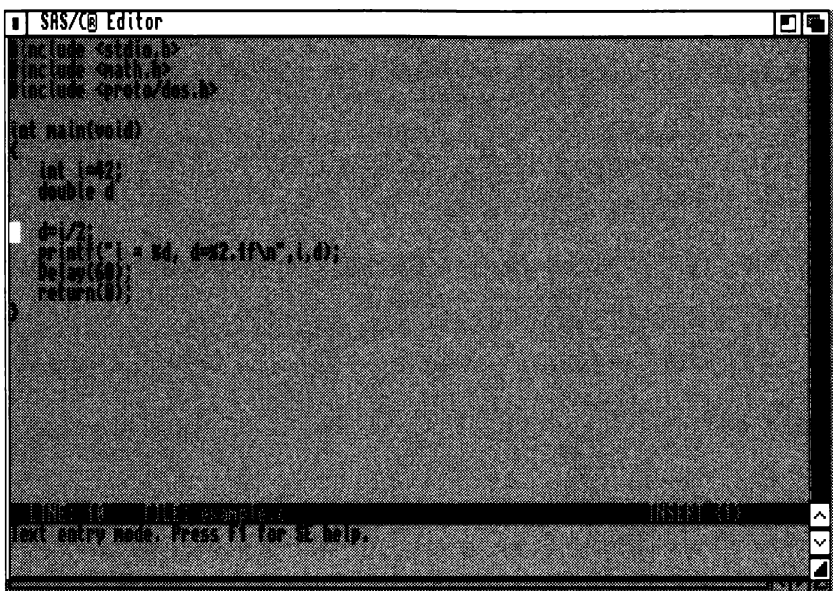
When the compiler finds the missing semicolon in your program, the compiler invokes `scmsg`, the message browser. `scmsg` opens a window in which it displays each of the messages generated by the compiler. Display 2.2 shows the message browser window generated when you compile `example.c`.

Display 2.2
*The SAS/C Message
Browser Window*



Double-click on the message in the message browser window, and the message browser invokes the editor on the file and line number specified in the message, as shown in Display 2.3.

Display 2.3
The Edit Window
Showing `example.c`



```

SAS/C Editor
#include <stdio.h>
#include <math.h>
#include <proto/dos.h>

int main(void)
{
    int i=42;
    double d;

    d=1/7;
    printf("i = %d, d=%2.1f\n", i, d);
    delay(40);
    return(0);
}

text entry mode. Press F1 for E help.

```

Correct the error by inserting a semicolon at the end of the declaration for `d`. Save the corrected file by selecting the **Save & Close** option from the **Project** menu or by pressing **Right Amiga-S**.

Close the message browser window by clicking on the **Close** gadget, and then close the **smake** window by pressing **Return**.

Recompile and link your program (also referred to as *rebuilding* your program) by double-clicking on the **Build** icon. As before, **smake** invokes the **sc** command to recompile your file. **smake** then invokes the linker, **slink**, to produce the executable module, **example**. The linker also creates an icon for the executable module. (If you are running under AmigaDOS 1.3, you need to close then reopen the project drawer to see the icon for **example**.)

For more information about the **scmsg** and **smake** utilities, refer to *SAS/C Development System User's Guide, Volume 2*.

Running Your Program

The build process produces an executable module and creates an icon for this module in the project directory. If the icon for this module is not displayed, close and reopen the window for the directory in which you created the file. To run **example**, double-click on the **example** icon.

When you run a program from the Workbench screen, the compiler opens a standard I/O window in which your program can display output. In the case of the **example** program, the output is as follows:

```
i = 42, d=21.00
```

If you are running under AmigaDOS 1.3, this window closes as soon as your program is finished running. For short programs, this window opens and closes so fast that it is difficult to see what is displayed in it. To make the window stay open for at least a second, the **example** program calls the AmigaDOS function **Delay**. If you are running under AmigaDOS 2.0, this window remains open until you click on its **Close** gadget. For information on changing the attributes of this window, see Chapter 9, “Running Your Program from the Workbench Screen.”

Debugging Your Program

When you compile your program with the **debug** option, you can use the CodeProbe debugger to monitor the behavior of your program, line-by-line, as it executes.

To run the debugger with the **example** file, click once on the **Debug** icon, then hold down the Shift key and double-click on the icon for the executable module. As shown in Display 2.4, CodeProbe opens two windows: the Source window that contains the source code for your program and the Dialog window into which you can enter CodeProbe commands.

Display 2.4
CodeProbe Windows
with `example.c`

```

1: #include <stdio.h>
2: #include <math.h>
3: #include <proto/dos.h>
4:
5: int main(void)
6: {
7:     int i=42;
8:     double d;
9:
10:    d=i/2;
11:    printf("i = %d, d=%2.1f\n", i, d);
12:    delay(60);
13:    return(0);
14: }

```

Task: test:cstuff/example at 0708C100

SAS/CPA Source Debugger V2.06.03
Copyright (c) 1988-1992 SAS Institute Inc.
example:\example.c:main 7

CodeProbe stops on the first line of your main program.

To step through your program one line at a time, you can press the Return key. Pressing the Return key is equivalent to entering the `proceed` command. The current (highlighted) line is the line that will execute next if you press Return. Press Return until the line containing `printf` is the current line.

To display the value of a variable, you can use the `display` command, which you can abbreviate as `d`. To display the value of the variable `d`, enter `d d` on the command line of the Dialog window. If you have already stepped through the line containing `d=i/2;`, then the debugger displays 21 as the value of `d`.

To finish executing your program, enter `g` (for `go`) on the command line of the Dialog window. You should always finish executing your program before quitting CodeProbe, or your machine may be left in an uncertain state and may crash later for no reason. To exit from the debugger, enter `q` (for `quit`).

The Debugger has considerably more capabilities than those described here. For a complete description of the CodeProbe debugger, refer to *SAS/C Development System User's Guide, Volume 2*.

Using SAS/C Tools from the Shell

The following sections describe how to use the SAS/C Development System from the Shell. Specifically, these sections discuss the following:

- creating a C source file using the **se** editor
- setting compiler options for the project using **scpts**
- compiling and linking a program using the **sc** command
- running your program
- using the debugger to look at your program.

If you are doing all of your work from the Shell, you do not need to use **scsetup** to set up a directory for your C files. However, you should create a new working directory using the AmigaDOS **makedir** command. It is recommended that you do not place your source code in the **sc** directory tree. Installing updates to the SAS/C Development System is easier if you can replace the entire directory without worrying about losing any of your own program files.

Creating a Source File with **se**

After you have created a directory for your C files, you can start entering your C programs. To create a new file, enter the **se** command, as follows:

```
se
```

se displays a window into which you can enter your program.

The **se** editor works the same whether you start it from the Workbench screen or from the Shell. See “Creating a Source File with **se**” under the previous section, “Using SAS/C Tools from the Workbench Screen,” for a description of entering a sample program.

Of course, after you save the file and close the window, icons are not displayed. You are prompted for another command.

For more information about the **se** editor, see Chapters 4 and 5 in Part 2, “Using the Screen Editor (se).”

Compiling, Linking, and Running Your Program

After you have created a C source program using the editor, you can compile, link, and run your program. If you want to compile your program with the default compiler options, simply enter the **sc** command followed by the name of the source file. However, to successfully run the examples shown in this chapter, you need to set additional compiler options. From the Shell, you can set compiler options using the **scpts** utility, or you can specify the compiler options on the **sc** command line. The next section, “Setting Compiler Options with **scpts**,” describes two different ways to use the set options from the Shell. The section

“Compiling and Linking with sc” describes how to specify options on the **sc** command line.

Setting Compiler Options with **scopts**

Usually, you compile all of the programs in a project with the same compiler options. The **scopts** utility allows you to set and save the options with which you want to compile the programs in a project.

From the Shell, you can run **scopts** in two different ways:

- You can enter the **scopts** command followed by the options with which you want to compile. For the example programs in this chapter, you should enter the following command:

```
scopts math=standard debug=symbolflush errorrex link
```

You can enter the options in uppercase, lowercase, or in mixed case. When you specify options on the **scopts** command, the **scopts** utility does the following:

1. reads the **scoptions** file in the current directory, if it exists
 2. adds or changes the options, depending on the options you specify on the **scopts** command
 3. writes the new option settings to the **scoptions** file in the current directory.
- You can enter the **scopts** command without specifying options and use the **scopts** screens to set the options:

```
scopts
```

scopts displays the first of several windows on which you can select the options with which you want to compile. For the example program in this chapter, you should select the **math=standard**, **link**, **errorrex**, and **debug=symbolflush** options as described in “Setting Compiler Options with **scopts**” under the previous section, “Using SAS/C Tools from the Workbench Screen.”

The **scopts** utility saves the options you select in the file **scoptions** in the project directory. When you compile your program, the compiler reads the options from **scoptions**.

For more information about the **scopts** utility, refer to *SAS/C Development System User’s Guide, Volume 2*.

Compiling and Linking with **sc**

If you chose to use **scopts** to set compiler options, then you can compile and link your **example.c** program by entering the **sc** command as follows:

```
sc example.c
```

sc reads the options from the **scoptions** file and compiles your program.

However, if you chose not to use **scopts** to set compiler options, you must enter the following command to compile and link the **example.c** program:

```
sc math=standard debug=symbolflush errorrexx link example.c
```

In either case, you specified the **link** option, so the compiler calls **slink** to link your program using the correct libraries. If your program links correctly, the linker creates an executable module named **example**.

When the compiler finds the missing semicolon in your program, the compiler invokes **scmsg**, the message browser. **scmsg** opens a window in which it displays each of the messages generated by the compiler. Display 2.2, earlier in this chapter, shows the message browser window generated when you compile **example.c**. (However, Display 2.2 shows the **snake** window behind the message browser window. On your machine, you will see a Shell window.)

Double-click on the message in the message browser window. The message browser invokes the editor on the file and line number specified in the message, as shown in Display 2.3, also shown earlier in this chapter.

Correct the error by inserting a semicolon at the end of the declaration for **d**. Save the corrected file by selecting the Save & Close option from the Project menu or by pressing Right Amiga-S. Close the message browser window by clicking on the **Close** gadget.

Recompile and link your program by entering the **sc** command as before. **slink** produces the executable module **example**.

For more information about the **sc** command, see Chapter 8, "Compiling and Linking Your Program."

Running Your Program

To run a program, enter the program name at the AmigaDOS prompt. To run **example**, enter the following:

```
example
```

The **example** program displays the following output:

```
i = 42, d=21.00
```

Debugging Your Program

When you compile your program with the **debug** option, you can use the CodeProbe debugger to monitor the behavior of your program, line-by-line, as it executes.

To run the debugger with the **example** file, enter the following command at the Shell prompt:

```
cpr example
```

As shown in Display 2.4, CodeProbe opens two windows: the Source window that contains the source code for your program and the Dialog window into which you can enter CodeProbe commands.

Enter debugger commands as described in “Debugging Your Program” under the previous section, “Using SAS/C Tools from the Workbench Screen.”

To finish executing your program, enter **g** (for **go**) on the command line of the Dialog window. You should always finish executing your program before quitting CodeProbe, or your machine may be left in an uncertain state and may crash later for no reason. To exit from the debugger, enter **q** (for **quit**).

The Debugger has considerably more capabilities than those described here. For a complete description of the CodeProbe debugger, refer to *SAS/C Development System User’s Guide, Volume 2*.

Using Icons

The tools included with the SAS/C Development System produce icons for most files that they generate. The **sc:icons** drawer contains template icons used by these tools. The icons in this drawer all have names starting with **def_**. With the exception of **def_exe** and **def_se**, the letter or letters after the **def_** represent the file extension of the files for which the icon is intended. For example, the **def_c** icon is the icon for files that end in **.c** (C source files), and the **def_h** icon is the icon for files that end in **.h** (C header files). If a file does not have an extension (that is, the filename does not contain a period), the entire filename is used as an extension. Therefore, you can create a default icon for **smakefile** files called **def_smakefile**. If a SAS/C tool needs to create an icon, it copies the icon matching the file’s extension from **sc:icons** to the correct directory and renames the icon to match the new filename.

If you save a file for which the editor cannot find an icon in **sc:icons**, the editor uses the default icon **def_se**. Whenever the linker creates icons for executable modules (which do not have filename extensions), it uses the icon **def_exe**.

The following table lists, for each tool that creates icons, the different types of items for which they create icons and the default icon each tool uses.

Command	Item	Icon
sc	listing files	def_lst
	object files	def_o
	preprocessor output	def_p
	generated prototypes	def_h
	GST files	def_gst
slink	executable programs	def_exe
	map files	def_map
se	any saved file	def_se is used if the correct icon does not exist
scsetup	creates icons for any file with an extension for which an icon exists in sc:icons .	

By default, the SAS/C Development System does not create icons for object (.o) files. However, the compiler and assembler check for an icon called **def_o** when they create an object file. If you want to see icons for object files, create an icon named **def_o.info** using any icon editor and save it to the **sc:icons** drawer under the name **def_o**.

To prevent all of the SAS/C commands from creating icons, rename the **sc:icons** drawer to **sc:noicons** or to any other name you like. If a tool cannot find the icons drawer, it does not create an icon.

To prevent a specific type of icon from appearing, rename or delete the template for that icon from the **sc:icons** drawer. For example, if you do not want your header (.h) files to have icons, rename or delete the **def_h** icon.

You can modify the icons in the **sc:icons** drawer in any way you choose. For example, you could replace the provided imagery with your own art work.

For additional information on using icons, refer to the descriptions of the Information option or Information window in *Using the System Software* (Commodore-Amiga, Inc. 1990).

3 Getting Help

27	<i>Introduction</i>
27	<i>Using the Help System</i>
29	<i>Contacting the Technical Support Division</i>
29	<i>Deciding When to Call Technical Support</i>
30	<i>Contacting Technical Support by Phone, Mail, or Fax</i>
31	<i>Contacting Technical Support through BIX</i>
31	<i>Contacting Technical Support through Internet or Usenet</i>
33	<i>Collecting the Information Needed by Technical Support</i>

Introduction

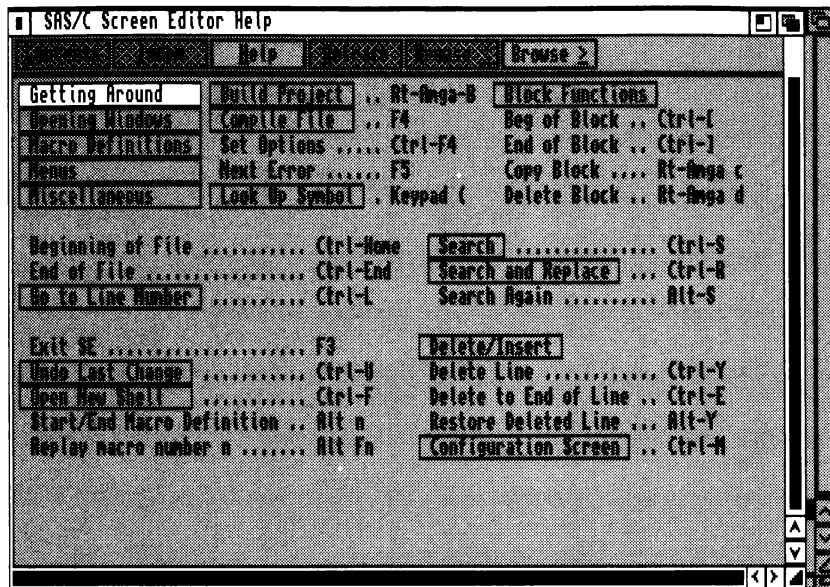
This chapter describes how to get help with any of the features of the SAS/C Development System. You have three primary sources of help:

- Appendix 1, “Solving Common Problems.” Appendix 1 describes solutions to the questions that users ask most frequently when they contact the Technical Support Division. This appendix also includes Questions that the Technical Support Division anticipates will be generated by the conversion to Version 6.
- The online help system. This system describes how to use each of the features provided by the SAS/C Development System.
- The Technical Support Division. You can contact the Technical Support Division if, for example, the compiler generates bad object code or error messages for correct code. Read the section “Contacting the Technical Support Division,” later in this chapter, before contacting the Technical Support Division.

Using the Help System

The SAS/C Development System provides an extensive online help facility using the AmigaGuide hypertext system provided by Commodore. From within `se`, `scopts`, `scmsg`, and CodeProbe, you can display help information by pressing the Help key. Within CodeProbe, you can also enter the `help` command on the command line. Display 3.1 shows the help screen that is displayed when you press the Help key while inside `se`.

Display 3.1
Editor Help Screen



You can also display help information by invoking the AmigaGuide system directly with the following command:

```
amigaguide database-name
```

The help databases are located in the directory **sc:help**. You can specify one of the following databases:

scmsg.guide

describes the error and warning messages returned by the compiler. When you press the Help key from within **scmsg**, you are accessing this database.

cpr.guide

describes the purpose of and parameters accepted by each of the CodeProbe commands. When you press the Help key from within CodeProbe, you are accessing this database.

se.guide

describes how to use **se**. When you press the Help key from within the editor, you are accessing this database.

sc_util.guide

describes how to use each of the utilities provided with the SAS/C Development System.

sc.guide

describes each of the options you can specify in the **sc** command.

When you press the Help key from within **scopts**, you are accessing this database.

sc_lib.guide

describes each of the library functions. This database also allows you to display lists of functions by category or view an alphabetical list of all functions.

To navigate through a database, simply click on any of the buttons that appear on a screen or select an option from the menu bar. The menu bar on all of the help screens contains the following options:

- | | |
|----------|---|
| Contents | displays the main help screen for the database that you are viewing. |
| Index | is a standard AmigaGuide option that is not used by the SAS/C Development System. |
| Help | displays information about using the help facility. |
| Retrace | backtracks through the help screens that you have already displayed. |
| Browse < | displays the previous screen in the database. |
| Browse > | displays the next screen in the database. |

To exit the help system, click on the **Close** gadget.

Contacting the Technical Support Division

When you purchased the SAS/C Development System, you received access to the SAS Institute Technical Support Division. Technical Support can help you with unexpected problems that might arise with the compiler. Technical Support also gathers users' comments on the current product and their suggestions for enhancements.

Deciding When to Call Technical Support

Before you call the Technical Support Division, read through Appendix 1, "Solving Common Problems." This appendix describes the solutions to several frequently encountered problems.

If you cannot find the answer to your problem in Appendix 1, you may need to call Technical Support. The following list describes some of the more common situations for which you should call Technical Support:

- ☐ The compiler generates an error for code that is correct.
- ☐ The compiler does not generate an error or warning for code that is incorrect.

- ☐ The compiler generates bad object code.
- ☐ The compiler detects an internal error (CXERR) condition.
- ☐ The compiler or any utilities crash or end abnormally.
- ☐ The library functions or utilities do not perform as specified.
- ☐ The manuals contain errors or do not provide enough information.

**Contacting
Technical
Support by
Phone, Mail,
or Fax**

If you live in the United States, Europe, or Canada, you can contact the Technical Support Division in writing, by phone, or by fax:

Address: SAS Institute Inc.
ATTN: Amiga SAS/C Technical Support
SAS Campus Drive
Cary, North Carolina 27513
USA

Phone: 919-677-8009

Fax: 919-677-8123

If you live in Australia, you should contact our Australian office:

Address: SAS Institute Australia Pty. Ltd.
ATTN: Amiga SAS/C Technical Support
Private Bag No. 52
Lane Cove, NSW 2066

Phone: (02) 428 0428

Fax: (02) 418 7211

If you live in New Zealand, you should contact our New Zealand office:

Address: SAS Institute (NZ) Ltd.
ATTN: Amiga SAS/C Technical Support
PO Box 10-109
The Terrace
Wellington

Phone: (04) 727 595

Fax: (04) 727 055

Whether you choose to call, send a letter, or send a fax, make sure you include or have with you all of the information described in the section “Collecting the Information Needed by Technical Support.”

Contacting Technical Support through BIX

BIX (Byte Information Exchange) is a subscription service bulletin board system available to users in Canada and the Continental United States. You can contact BIX at the following address:

Address: BIX/General Videotex Corporation
1030 Massachusetts Avenue
Cambridge, MA 02138
USA

Phone: 1-800-695-4775 (outside Massachusetts)
617-354-4137 (inside Massachusetts)

You can sign up for a BIX subscription as follows:

1. Through a modem, dial BIX at 1-800-225-4129.
2. At the `login:` prompt, enter `bix`.
3. At the `Name?` prompt, enter `bix.amiga`.

When you subscribe to BIX, you receive a user's manual that covers all aspects of their services, including:

- ☐ the various conferences available
- ☐ how to access a conference
- ☐ how to access source listings.

SAS Institute maintains a support conference on BIX named `sas.c`. This conference is monitored Monday through Friday for new messages. You can contact Technical Support by posting messages to this conference.

Note: Do not post your registration number to the public conference.

You can also send private email messages on BIX to the ID `sas.c` (the same name as the conference).

Whether you choose to post to the public conference `sas.c` or to send a private message, make sure you include all of the information described in the section “Collecting the Information Needed by Technical Support.”

Contacting Technical Support through Internet or Usenet

You can also contact the Technical Support Division through Internet using the EMITS (Electronic Mail Interface to Technical Support) facility.

EMITS logs your mail for the attention for the Technical Support Division and sends an email message acknowledging receipt of your mail.

Before you can use this facility, you must register with EMITS. If you have accounts on multiple machines, register from the same host system that you want to use for sending messages to the Technical Support Division. EMITS extracts your email address from the header of your message and uses this address as your account ID. You may register from multiple machines if you want to send messages to Technical Support from more than one machine. Register with EMITS by sending the following information to **support@sas.com**:

```
./register=your name
./site=registration number
./company=company name or your name (if registering a private copy)
./phone=phone number
```

Include the word **EMITS** in the subject of the message. Begin the **./** in column 1.

Mail that does not include all of this information or that is syntactically incorrect is returned to you.

Here are some examples:

```
./register=John Doe
./site=SAS-000001
./company=Jane Doe
./phone=(555)555-5555
```

Note: Your phone number does not need to conform to this syntax. You can specify phone numbers in any form necessary.

When your account has been added, EMITS notifies you by email and sends you a complete guide to using EMITS.

There are several points to keep in mind when sending mail to EMITS.

- ☐ All email messages to EMITS should be sent to **support@sas.com**.
- ☐ All email messages intended for EMITS must contain the words **EMITS** in the subject.
- ☐ EMITS is not case sensitive.
- ☐ EMITS can process only one request for each email message you send. For example, send two separate email messages to **support@sas.com** to register and to request the help file.

When you contact Technical Support through EMITS, make sure you include in your message all of the information described in the next section.

Collecting the Information Needed by Technical Support

The Technical Support staff requires specific information before they can handle your questions. Include the following information in your correspondence, or have this information ready if you call:

Registration number

Technical Support cannot answer any questions unless you provide a valid registration number. This number is found on a cardboard sheet included with the compiler documentation. The postcard that is part of that sheet should be torn loose and mailed to SAS Institute as soon as possible after purchasing the compiler. Please keep the other portion of that sheet with your documentation. If you have upgraded the compiler from an older, registered release, you do not need to send in the new card. If you decide to send in the new card, please note your old number, so that it may be removed from our database. If you do not know your registration number, please contact Book Sales at (919) 677-8000, extension 5042.

If you purchased your copy of the compiler from someone who has already registered with us, please contact Book Sales with the following information:

- ☐ the name, address, phone number, and registration number for the previous owner, so that their name may be removed from our database.
- ☐ your name, address, phone number, and registration number so that you may be added to the database and become eligible for Technical Support.

Version number

The version number of the compiler is displayed in the header that appears on the screen each time you run the compiler. You can also display the version number with the AmigaDOS `version` command.

Return address and phone number

If you send a letter or fax, include your return address in the correspondence — not just on the envelope. Whether you call or write, Technical Support will need a phone number where you can be reached from 9:00 am to 5:00 pm EST.

Complete problem description

Include all error messages and symptoms of the problem. If you are not able to call when you have access to your machine, write down the following information before contacting Technical Support:

- ☐ the exact text of any error messages, guru meditation numbers, and so on, that you see on the screen
- ☐ whether the problem occurs when you are compiling, linking, or executing your program

- any commands, options, or code needed to consistently reproduce the problem (for example, compiler or linker options and CodeProbe commands).
- any parts of the product that you chose not to install.
- the configuration of your machine, including the type of processor, the amount of memory, and the version of the AmigaDOS operating system.

If you have problems with a large piece of code, try to isolate the code that is causing the problem into a test case of 30 lines or less.

Technical Support staff can respond faster if you can reduce the problem to a small test case.

■ Part 2

Using the Screen Editor (se)

Chapters	4	Using se
	5	Controlling se
	6	Customizing the Editor
	7	Using the AREXX Interface to se

4 Using **se**

- 37 *Introduction*
- 38 *Starting the Editor*
- 39 *Reading the Status Line*
- 40 *Moving Around in a File*
- 40 *Entering and Editing Text*
 - 41 *Searching for and Replacing Character Strings*
 - 43 *Working with Blocks of Text*
 - 44 *Undoing Your Last Change*
 - 45 *Using Keystroke Macros*
 - 46 *Entering Control Characters*
- 46 *Saving Your File and Exiting **se***
- 47 *Managing Windows*
- 48 *Compiling from within the Editor*
- 48 *Getting Help*

Introduction

This chapter describes how to use the editor, **se**. Specifically, this chapter describes how to:

- ☐ invoke the editor
- ☐ insert text into and delete text from a file
- ☐ search for and replace character strings
- ☐ create and use keystroke macros
- ☐ save your file and exit the editor
- ☐ open, close, and switch between windows
- ☐ compile your program from within **se**
- ☐ get help from within **se**.

This chapter describes the most commonly used features of **se**. It does not cover all of the commands available or all of the ways in which you can execute commands. Chapter 5, “Controlling **se**,” contains comprehensive lists of the commands available in **se**.

Note: The instructions in this chapter assume that you are using the default keystroke definitions. If you have customized your key definitions with the **se** configuration window, use your customized keystrokes when required.

Starting the Editor

You can start **se** by:

- double-clicking on the **Edit** icon.
- entering the **se** command at the Shell prompt.

The **se** command has the following format:

se [*option*] [*filename*]...

where *option* is one of the following:

-v

displays the **se** version number and copyright statement and exits **se**. If you specify the **-v** option, **se** ignores anything else you specify in the **se** command. You need the version number from this display if you call the Technical Support Division for help with **se**.

-dfilename

uses the data file that you specify instead of **se.dat**. This file contains keyboard definitions and default information, including color, window size, and so on. When you start the editor, **se** looks for a data file. By default, **se** looks for **se.dat** in **env:sc** and, if it does not find the file, **se** looks in **sc:env**. You can tell **se** to use a different data file by specifying the filename with the **-d** option.

If you start **se** by double-clicking on the **Edit** icon or by entering the **se** command without a filename, **se** displays an empty edit window.

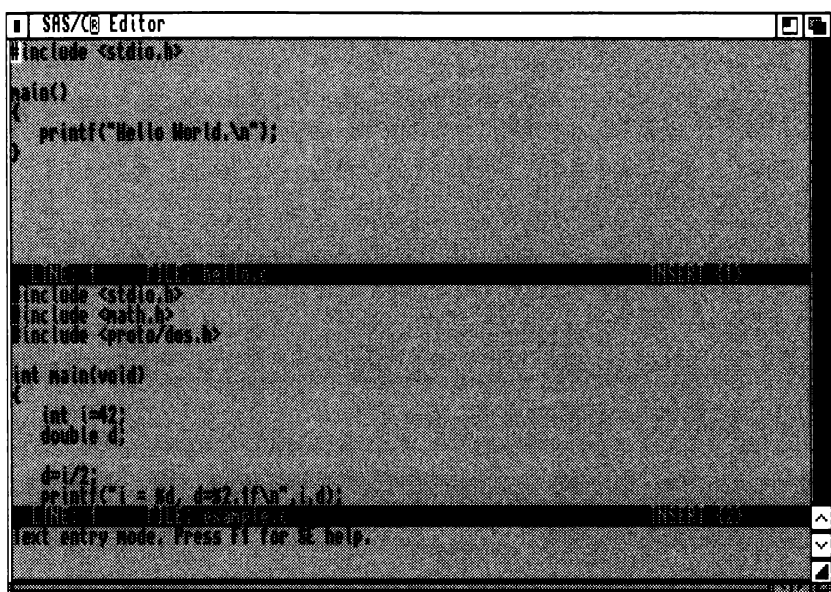
When you start **se** from the Shell prompt, you can specify the names of the files that you want to edit. If you specify more than one filename, **se** opens a window for each file. **se** displays the first file in the top window and the second file in the bottom window. If you enter more than two filenames, you must press the F6 (cycle windows) key to display the additional files. You can specify up to nine filenames on the **se** command line.

For example, you can edit the files **hello.c** and **example.c** from the **sc/examples** directory by entering the following commands:

```
cd sc:examples
se hello.c example.c
```

Display 4.1 shows the two edit windows displayed by **se**.

Display 4.1
Editing `hello.c` and
`example.c`



At the bottom of each edit window is a status line that shows, among other things, the name of the file in that window. The last two lines on the screen are the lines that *se* uses to display messages and to prompt you for additional information.

You can have up to nine files open at one time. To open an additional file, select Open Window from the Windows pull-down menu. *se* asks you for the file that you want to open. If you are running AmigaDOS 2.0, *se* displays a file requester. Each open file is displayed in a different window, and you can switch between files by selecting Switch Windows from the Windows pull-down menu. The current window is the window that contains the cursor.

The default text window size is 20 lines by 78 columns. You can set each window to occupy a half screen or a full screen by selecting Toggle Display Size from the Windows pull-down menu.

Reading the Status Line

The status line contains the following items:

- ☐ current line number.
- ☐ current column number. If you turn on the Column Display option from the *se* configuration menu, *se* displays the current column number in the status line. Chapter 6, “Customizing the Editor,” describes how to use the *se* configuration menu.

- current filename.
- the current mode. You can switch between two modes, insert mode and overwrite mode, by pressing the Insert key (NUM-0). In insert mode, any character you enter is inserted at the current cursor position. In overwrite mode, any character you enter replaces the character at the current cursor position.
- window number. **se** displays the window number within braces. For example, for window number 2, **se** displays { 2 }.
- keystroke saver active indicator. By default, the Alt-*n* key sequence starts recording your keystrokes in that macro. On the status line, **se** displays the number *n* that you entered. For more information on creating keystroke macros, see “Using Keystroke Macros,” later in this chapter.
- marked block indicator. When you mark a block of text, **se** displays [*] in the status line. The section “Working with Blocks of Text,” later in this chapter describes how to mark a block of text.

Moving Around in a File

In addition to the arrow keys, you can use the following keys to move around in your file:

Home (NUM-7)

moves the cursor to the beginning of the current line

End (NUM-1)

moves the cursor to the end of the current line

Control-Home

moves the cursor to the beginning of the file

Control-End

moves the cursor to the end of the file

PgUp (NUM-9)

displays the previous page of text

PgDn (NUM-3)

displays the next page of text.

Entering and Editing Text

The following paragraphs describe the most common commands used to insert and delete characters. **se** offers many more commands that are described in Chapter 5, “Controlling se.”

When you first open your file, you are in insert mode. Any character that you enter is inserted at the current cursor position. If you press the

Insert key, **se** switches to overwrite mode, and any character you enter replaces the character at the current cursor position. Each time you press the Insert key, you switch modes.

As stated previously, the default window width is 78 columns. The maximum line length in **se** is 256 characters. If you have not selected Autowrap from the Options pull-down menu in the **se** configuration menu, then **se** does not automatically start a new line when you type past column 78. Instead, **se** moves the text to the left so that you can still see what you are typing. When you press Return, **se** redisplay the window beginning with column 1 and displays a greater than (>) sign at the right side of the screen on the line that exceeds 78 columns. (For more information about the Options pull-down menu, see Chapter 6, “Customizing the Editor.”)

You can use the following keys to delete text:

Backspace	deletes the character to the left of the cursor
Delete	deletes the character at the cursor position
Control-Y	deletes the current line
Control-E	deletes all characters from the current cursor position to the end of the current line.

If you decide you want to restore the last line that you deleted with Control-Y, press Alt-Y.

Searching for and Replacing Character Strings

The following sections describe the Search command and the Search and Replace command. To use these commands, you can enter strings as regular expressions or as simple character strings. By default, **se** expects you to enter regular expressions. *Regular expressions* are character strings that may contain characters that have special meanings. For example, **abcd**, **a little lamb**, and **[tT]+he\$** are all valid regular expressions. “Entering Regular Expressions,” later in this chapter, provides additional information on using regular expressions under **se**.

If you select String Matching as the Searches Method in the **se** configuration menu, then **se** interprets any special regular expression symbols as plain text.

Searching for Character Strings

To search for a character string, press Right Amiga-F. **se** asks you to enter the string for which you want to search:

```
REGULAR EXPRESSION SEARCH STRING: (MENU KEY to search backwards)
```

To search forward (from the current position toward the end) through the file, type the character string and press Return.

Note: The Search and Search and Replace commands are case-sensitive. Enter the character string exactly as it appears in the file.

To search backward (from the current position toward the beginning) through the file, press the F2 key, and **se** displays the following prompt:

(BACKWARDS) REGULAR EXPRESSION SEARCH STRING:

Type the character string and press Return.

After you press Return, **se** positions the cursor at the first occurrence of the character string. To repeat the search, press Right Amiga-A.

Replacing Character Strings

To replace one character string with another character string, press Right Amiga-R. **se** displays the same prompt as if you had pressed Right Amiga-F. You can type the character string that you want to replace and press Return, or you can press the F2 key if you want to search backward through the file. If you press F2, **se** displays the same prompt as if you were simply searching for the string, and you should type the character string that you want to replace and press Return.

After you enter the character string for which you want to search, **se** prompts you for a second character string:

REPLACE STRING:

Type the string with which you want to replace the first character string and press Return. **se** then asks you if you want to be prompted before it replaces any occurrences of the first string with the second string:

REPLACE: Prompt No Prompt

If you want **se** to ask you before it replaces any strings, select Prompt. (Use the arrow keys to highlight Prompt and press Return, or press the letter **p**.)

If you select No Prompt, **se** replaces all occurrences of the first string with the second string from the current position through the remainder of the file. (That is, to the end of the file for a forward search or to the beginning of the file for a backward search).

If you select Prompt, **se** positions the cursor at the first occurrence of the character string, and asks you what you want to do:

REPLACE: Yes No Quit Global

Select one of the options:

- Yes replaces the string and positions the cursor at the next occurrence of the character string.
- No moves the cursor to the next occurrence of the character string.
- Quit stops the search and replace without replacing any strings.
- Global replaces all remaining occurrences of the string without displaying any prompts.

You can highlight your selection by pressing the arrow keys or by pressing the first letter of the option.

Entering Regular Expressions

As previously stated, regular expressions are character strings that may contain characters that have special meanings. Regular expressions are also referred to as *patterns*.

Refer to the section “Specifying the Pattern” in the description of the **grep** utility in *SAS/C Development System User’s Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0* for a complete description of entering regular expressions. **se** supports regular expressions in the same manner as **grep** except that you do not need to enclose in double quotes (“”) a string that contains spaces. For example, to search for the string *a little lamb*, you would enter `a little lamb` instead of `"a little lamb"`.

Working with Blocks of Text

You can copy, delete, move, print, and write entire blocks of text within the same file, between different files, or to and from the clipboard. To perform an operation on a block of text, you must first mark the block.

You can mark a block of text in one of two ways:

- ☐ Position your cursor on the first character that you want included in the block, and press Control-[. **se** displays a left bracket in the status line. Then, position your cursor after the last character that you want included in the block, and press Control-].
- ☐ Position your mouse cursor on the first character that you want included in the block, and press and hold down the left mouse button. Move the mouse cursor to the last character that you want included in the block, and release the mouse button.

se displays the marked block in reverse video and displays `[*]` in the status line. You can have only one marked block at a time.

After you have marked a block, you can delete or print the block by selecting the appropriate option from the Block pull-down menu. You can move or copy the block to another open file by positioning the cursor at

the location to which you want to move or copy the block and selecting the appropriate option from the Block pull-down menu. You can also use the keyboard shortcuts displayed beside the menu selection.

To write the block to a file that is not open, select the Write option from the Block pull-down menu. **se** asks you to enter the name of the file to which you want to write the text:

```
WRITE BLOCK TO FILE:  Filename
```

Type the appropriate filename and press Return. **se** writes the block to the filename you enter. The block text replaces any text that is already in the file.

If you mark a block of text and then decide that you do not want to perform any operations on the block, you can unmark the block by pressing Control-[, then Control-] without moving the cursor.

Note: The Read option in the Block pull-down menu performs the same function as the Insert File option in the Project menu.

Undoing Your Last Change

se remembers the changes you entered for the last 50 lines that you changed, inserted, deleted, and so on. You can undo these changes with the **undo** command. You can run the **undo** command in any of three ways:

- ☐ press Right Amiga-U
- ☐ press Control-U
- ☐ select Undo Last Change from the Project menu.

When you select the **undo** command, **se** displays a prompt depending on the last command you entered:

```
UNDO -- DELETE INSERTED LINE?:  Yes  No
UNDO -- DELETE ADDED BLOCK?:    Yes  No
UNDO -- RESTORE DELETED BLOCK?:  Yes  No
UNDO -- RESTORE DELETED LINE?:   Yes  No
UNDO -- RESTORE REPLACED LINE?:  Yes  No
```

Select the appropriate answer and press Return.

Note: **se** remembers changes for the last 50 lines only. If you delete or add a block of text larger than 50 lines, **se** remembers only the first 50 lines.

Using Keystroke Macros

If you need to perform repeatedly a task that requires several keystrokes, you may want to create a keystroke macro to do the task. If you save your keystrokes as a macro, then you can perform the task by pressing only the Alt key and a function key.

Creating and Replaying Keystroke Macros

To create a keystroke macro, hold down the Alt key and press a number (0-9) key (for example, Alt-0). The number that you press is displayed in the right side of the status line between the window number and the marked block indicator (if present). *se* displays the following message:

Key Saver Begun

Note: If you press 0, the status line shows the number 10.

From this point, all of your keystrokes are stored as part of the macro. Enter all of the keystrokes necessary to perform the task. After you have completed the task, press Alt and the same number key again to stop recording keystrokes. To execute, or replay, the macro, press Alt-Fn, where *n* is the number you pressed to define the macro (for example, Alt-F10).

Any macros you define are saved until you exit from *se*. The following section, “Saving and Reloading Keystroke Macros,” describes how to save keystroke macros in a file and reuse them later.

Saving and Reloading Keystroke Macros

If you do not save your keystroke macros in a file or convert them to a key as described under “Assigning a Macro to a Different Key,” they are lost when you complete your editing session and terminate *se*.

To save your keystroke macros in a file, select the Save Macros option from the Project pull-down menu. *se* prompts you for a filename in which to save the macros:

SAVE MACRO FILE: SPECIFY MACRO FILENAME (default is SE.MAC)

Type a filename and press the Return key.

To reuse your macros, select the Load Macros option from the Project pull-down menu. *se* prompts you for the name of the file in which you saved the macro definitions:

LOAD MACRO FILE: SPECIFY MACRO FILENAME (default is SE.MAC)

Type the filename and press the Return key. The file that you specify is loaded over any macro files that you may have already loaded. **se** does not check the file to determine if the file is a valid macro file.

Assigning a Macro to a Different Key

When you create a keystroke macro, that macro is assigned to one of the 10 possible Alt-Fn key sequences. If necessary, you can assign one or more of the macros to a different key sequence. For example, you may want to assign the Alt-F10 macro to Control-Z.

After you assign a macro to a key, you can edit and save the macro using the **se** configuration window.

To assign a keystroke macro to a different key sequence, select the Convert Macro to Key option from the Options pull-down menu. **se** asks you to enter the macro number:

Enter the macro number to convert to a key(1-10)

Type the appropriate number and press Return. To convert macro number 10, type the number 10, not just a 0 as you did when you created the macro.

se displays the keymap portion of the configuration window. Select the key sequence to which you want to assign the macro by clicking on or pressing the appropriate keys such as the Control key and the letter Z. After you have selected the keys, click on the **Close** gadget.

To save the macro assignment for your next editing session, you must redisplay the **se** configuration menu by selecting the Configuration Window option from the Options pull-down menu. Then, click on the **Save** gadget or select the Save As option from Project pull-down menu.

Entering Control Characters

You can enter control characters into your text file by preceding the control character with the **se** escape character, Control-\. For example, to enter a form feed character into your text file, press Control-\
Control-L.

Saving Your File and Exiting se

After you have finished editing your file, you can save the file, exit the file without saving your changes, and/or exit **se**.

You cannot save an unnamed file. If you started **se** by double-clicking on the **Edit** icon or by entering the **se** command without a filename, then you must name the file before you can save it. To name the file,

select the Rename option from the Project pull-down menu. *se* asks you to enter a filename:

ENTER NEW FILENAME for current file:

Type the appropriate filename and press Return. You can now save the file and quit *se*.

To save your file and/or quit the editor, select the appropriate option from the Project pull-down menu:

Save & Close

saves your file and closes the current window. If you have only one window open, this option also terminates *se*. If you have more than one window open, the next window becomes the current window.

Close Window

closes the current window. This option does not save your file. If you have only one window open, this option also terminates *se*. If you have more than one window open, the next window becomes the current window.

Save & Continue

saves your file. This option does not close any windows.

Save & ReOpen

saves and closes your file, then asks you for the name of a new file to edit. This option does not open an additional window to edit the new file.

Exit SE

terminates the editor. This option does not save your file.

You may also use the keyboard shortcuts displayed beside the menu options.

Managing Windows

In *se*, you can have up to nine files open at one time, and each open file is displayed in a different window.

To open an additional window, select the Open Window option from the Windows pull-down menu. *se* asks you for the name of the file to read into the new window:

OPEN WINDOW: Filename (Menu Key for view mode only)

Type the appropriate filename and press Return.

If you have only one window open, that window occupies the entire screen. By default, if you open two or more windows, **se** displays two windows at a time in split-screen mode. In split-screen mode, each window occupies half of the screen. Display 4.1, earlier in this chapter, shows two windows in split-screen mode.

In split-screen mode, you can move between the two currently displayed windows by pressing Control-F6. Within the same window, you can cycle through the remaining (hidden) windows by pressing F6.

To switch between split-screen mode and full-screen mode, select the Toggle Display Size option from the Windows pull-down menu. To display the names of each open file, select the Display Names option from the Project pull-down menu.

Compiling from within the Editor

Before you compile your program, make sure that you have the correct options specified in **scopts**. To specify compiler options from within **se**, press Control-F4 to run the **scopts** utility. If you change any settings, save the new settings before exiting **scopts**.

To compile your program, press F4. **se** displays a message telling you that your program is compiling. If the compiler finds any errors in your program, it sends the errors to the **scmsg** utility, and **scmsg** displays the errors in the Message Browser window. To locate the line causing an error, double-click on the error message. To move to the next error, press F5.

For additional information on using the **scmsg** and **scopts** utilities, refer to Chapter 2, “Using Your SAS/C Development System,” or to the description of the **scmsg** utility in *SAS/C Development System User’s Guide, Volume 2*.

Getting Help

To display help in **se**, press the Help key or press F1. **se** displays either the SAS/C Screen Editor Help menu or context-sensitive help information for the command you are executing. Chapter 3, “Getting Help,” describes how to use the help system.

5 Controlling se

49	<i>Introduction</i>
50	<i>Selecting Menu Options</i>
51	<i>Moving Around in the File</i>
52	<i>Inserting Text</i>
52	<i>Deleting Text</i>
53	<i>Working with Blocks of Text</i>
53	<i>Searching for and Replacing Strings</i>
54	<i>Managing Windows</i>
54	<i>Changing Colors</i>
55	<i>Creating and Using Keystroke Macros</i>
55	<i>Editing a New File</i>
56	<i>Naming Files</i>
56	<i>Undoing Changes</i>
56	<i>Saving Your File and Exiting the Editor</i>

Introduction

This chapter describes all of the commands available within **se** and how to invoke each command. Some of the commands listed in this chapter are described in more detail in Chapter 4, “Using se.”

Note: This chapter describes the default keystroke definitions. If you have customized your key definitions with the **se** configuration window, substitute your customized keystrokes when necessary.

The first section in this chapter, “Selecting Menu Options,” describes the different ways in which you can select options. The remaining sections describe the options and commands you can use to perform specific tasks:

- ☐ moving around in a file
- ☐ inserting text
- ☐ deleting text
- ☐ marking, copying, deleting, and moving blocks of text
- ☐ searching for and replacing character strings
- ☐ opening, closing, and changing windows
- ☐ creating and using keystroke macros
- ☐ editing a new file
- ☐ saving your file and exiting the editor.

Each section includes a table that shows, for each specific task, which keys/options to press/select to perform the task. For example, to move to a specific line in your file, you can:

- Press Control-L.
- Press Right Amiga-L.
- Select the Line Number option from the Search pull-down menu.

Selecting Menu Options

You can select an option or enter a command in one of three ways:

- Select an option from a pull-down menu.
- Enter the appropriate accelerator key for the option or command.
Many of the menu options are also associated with menu accelerator keys (also called *keyboard shortcuts*). These shortcuts are displayed to the right of the option in the pull-down menus. For example, the keyboard shortcut for the Save & Close option in the Project menu is Right Amiga-S.
- Select an option from the abbreviated menus displayed with the F2 (Main Menu) key. If you press the F2 key, **se** displays an abbreviated menu of the most frequently used commands at the bottom of the screen:

COMMAND: B C F M L O P R S Q U

Each of these options either displays a submenu or executes a command. For example, B displays the Block menu, and Q exits the editor. To display a brief description of an option, move the cursor to that option.

se also provides context-sensitive help that describes each command in detail. To display help information, press the Help key.

Moving Around in the File

The following table lists the keys/options that you can use to move around in your file.

Task	Key	Menu/Option
Move up one line	Up arrow	
Move down one line	Down arrow	
Move left one character	Left arrow	
Move left one word	Control-Left arrow Shift-Left arrow	
Move right one character	Right arrow	
Move right one word	Control-Right arrow Shift-Right arrow	
Go to a specific line	Control-L Right Amiga-L	Search/Line Number
Go to the beginning of the line	Home	
Go to the end of the line	End	
Go to the top of the file	Control-Home	
Go to the end of the file	Control-End	
Move up one page	PgUp	
Move down one page	PgDn	
Scroll the text up	Control-6	
Scroll the text down	Control-V	
Go to the beginning of a marked block		Block/Beginning
Go to the end of a marked block		Block/End

Inserting Text

The following table lists the keys and options that you can use to insert text into your file.

Task	Key	Menu/Option
Switch between Insert and Overwrite modes	Ins	
Insert a line before current line	Control-I	
Restore deleted line	Alt-Y	
Enter control characters	Control-\c	
Insert a file at current position		Project/Insert File

Deleting Text

The following table lists the keys/options that you can use to delete text from your file.

Task	Key	Menu/Option
Delete current character	Del	
Delete previous character	Backspace	
Delete word	Control-W	
Delete line	Control-Y	
Delete from cursor position to end of line	Control-E	
Delete a marked block	Right Amiga-D	Block/Delete

Working with Blocks of Text

The following table lists the keys/options that you can use to mark, copy, move, insert, and delete blocks of text. For a more detailed description of these commands, see Chapter 4, “Using se.”

Task	Key	Menu/Option
Mark beginning of block	Control-[	
Mark end of block	Control-]	
Copy a marked block	Right Amiga-C	Block/Copy
Delete a marked block	Right Amiga-D	Block/Delete
Move a marked block	Right Amiga-M	Block/Move
Print contents of a marked block		Block/Print
Read a disk file into the editor window		Block/Read
Write contents of marked block to a file		Block/Write
Go to the beginning of a marked block		Block/Beginning
Go to the end of a marked block		Block/End

Searching for and Replacing Strings

The following table lists the keys/options that you can use to locate character strings and to replace a string with another string. For a more detailed description of these commands, see Chapter 4, “Using se.”

Task	Key	Menu/Option
Search for a character string	Control-S Right Amiga-F	Search/String Search
Replace one character string with another	Control-R Right Amiga-R	Search/Search and Replace
Repeat last search or replace command	Alt-S Right Amiga-A	Search/Search Again

Managing Windows

The following table lists the keys/options that you can use to open, close, and cycle through windows. Some of these commands are described in more detail in Chapter 4, “Using se.”

Task	Key	Menu/Option
Switch window (in split-screen mode)	Control-F6 NUM-5	Windows/Switch Windows
Cycle windows (in full-screen mode)	F6	
Change window size	F9	Windows/Toggle Display Size
Switch between 20 lines and 45 lines	NUM Enter Right Amiga-H	Windows/Interlace Toggle
Open new window	Control-O	Windows/Open Window
Open a new Shell	Control-F	Windows/Create New CLI
Move window front to back	F10	
Close a window	Right Amiga-Q	Project/Close Window

Changing Colors

The colors used in the **se** windows are based on your Workbench colors. You can change the way the Workbench colors are used in the **se** windows with the keys listed in the following table. These keys allow you to cycle through the foreground and background colors.

Task	Key
Cycle foreground colors	F7
Cycle background colors	F8

Creating and Using Keystroke Macros

The following table lists the keys/options that you can use to create, save, and load keystroke macros. For a more detailed description of these commands, see Chapter 4, “Using se.”

Task	Key	Menu/Option
Start/stop saving keystrokes	Alt-n	
Execute a keystroke macro	Alt-Fn	
Save macros in a file		Project/Save Macros
Load macros from a file		Project/Load Macros
Assign keystroke macro to a different key		Options/Convert Macro to Key

Editing a New File

The following table lists the keys/options that you can use to open new files. For a more detailed description of these commands, see Chapter 4, “Using se.”

Task	Key	Menu/Option
Save current file and edit new file	Right Amiga-N	Project/Save & Reopen
Open a new file in the current window	Right Amiga-O	Project/Open New Window
Open a new file in a new window	Control-O	Windows/Open Window

Naming Files

The following table lists the keys/options that you can use to change the names of files from inside **se**. For a more detailed description of these commands, see Chapter 4, “Using **se**.”

Task	Menu/Option
Rename current file	Project/Rename
Display names of files being edited	Project/Display Names

Undoing Changes

The following table lists the keys/options that you can use to undo your last change.

Note: The action taken by the Undo Last Change option depends on the last command you entered. For a complete description of this option, see Chapter 4, “Using **se**.”

Task	Key	Menu/Option
Restore the last line you deleted	Alt-Y	
Undo last change	Control-U Right Amiga-U	Project/Undo Last Change

Saving Your File and Exiting the Editor

The following table lists the keys/options that you can use to save your file and exit the editor. For a more detailed description of these commands, see Chapter 4, “Using **se**.”

Task	Key	Menu/Option
Save the file in the current window	Right Amiga-W	Project/Save & Continue
Save the file and close the current window	Right Amiga-S	Project/Save & Close
Save current file and edit new file	Right Amiga-N	Project/Save & Reopen
Exit the editor without saving your file	F3 Right Amiga-X	Project/Exit SE

6 Customizing the Editor

57	<i>Introduction</i>
57	<i>Displaying the Configuration Window</i>
59	<i>Displaying Key Definitions</i>
60	<i>Setting Key Definitions</i>
61	<i>Setting Tab Stops</i>
61	<i>Setting Other Options</i>
64	<i>Saving the New Settings</i>
64	<i>Reusing Saved Settings</i>

Introduction

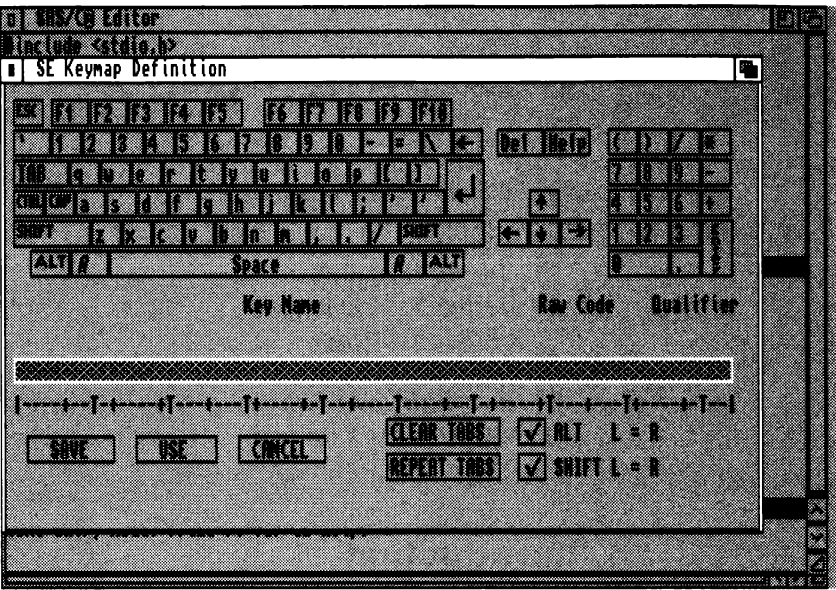
This chapter describes how to use the **se** configuration window. The configuration window allows you to set options and change the functions assigned to different keys and key combinations (*sequences*). For example, you can do the following:

- ☐ set the input processing mode
- ☐ set tab stops
- ☐ set automatic indention
- ☐ specify whether searches are to be simple string searches or pattern searches
- ☐ change the command assigned to a key or key combination.

Displaying the Configuration Window

To display the **se** configuration window shown in Display 6.1, open the Options pull-down menu on the **se** menu bar, and select the Configuration Window option. (Alternatively, you can press Control-M.)

Display 6.1
se Configuration Window



To display the configuration window menu bar, press and hold the right mouse button. The menu bar contains six pull-down menus:

- Project contains options that allow you to save any changes you make while in the configuration menu or to quit the configuration window without saving your changes. The sections “Saving the New Settings” and “Reusing Saved Settings,” later in this chapter, describe how to use the Project menu.
- Options allows you to specify several options that determine how the editor works. For example, you can specify how tab characters are processed, whether Search and Replace commands are case sensitive, whether the **se** status line shows the current column number, and so on. The section “Setting Other Options,” later in this chapter, describes how to use this option.
- Keys 1-4 lists the different commands that you can assign to a key or key combination. The following sections “Displaying Key Definitions” and “Setting Key Definitions” describe how to use these menus.

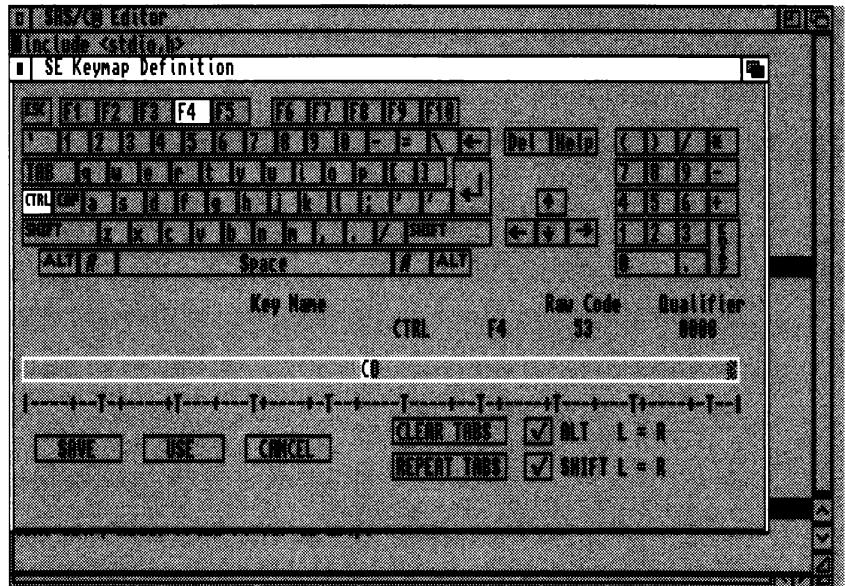
Displaying Key Definitions

To display the command assigned to a key or key sequence, you can

- ☐ click on that key or key sequence in the keymap.
- ☐ press that key, if the key is a qualifier key. *Qualifier keys* are keys that modify the function of other keys, such as the Control key, the Alt key, the Shift key, and so on.

For example, to display the definition of Control-F4 as shown in Display 6.2, click on the Ctrl key and the F4 key.

Display 6.2
se Configuration
Window Showing
Ctrl-F4



The name of the key or key sequence appears in the Key Name field, **CTRL F4**, and the command assigned to the key appears in the text gadget below the Key Name, **CO**.

To determine the function of a specific command, open the Keys pull-down menus. These menus contain the explanations of each of the commands that you can assign to a key or key sequence. For example, the Keys 2 pull-down menu shows the function of the **CO** command as Set SC Options.

The configuration screen also displays the raw code and qualifier associated with a specific key combination.

- Raw code** is a hexadecimal code associated with a specific key on the keyboard by the hardware. The AmigaDOS operating system uses keymaps to convert this raw code into a recognizable character.
- Qualifier** indicates which qualifier keys are part of a key sequence. As stated before, qualifier keys are keys that modify the function of other keys, such as the Control key, the Alt key, the Shift key, and so on.

The raw codes and qualifiers are displayed for reference only. For more information about raw codes and qualifiers, refer to the description of **keyboard.device** in *Amiga ROM Kernel Reference Manual: Devices, 3rd Edition* (Commodore-Amiga, Inc. 1991).

After you have displayed a key definition, you can press Return to unselect the key, clear the text gadget, and remain in the configuration program. You can also select **CANCEL** or click on the **Close** gadget to exit the configuration program.

Setting Key Definitions

By default, the left and right Alt keys are equivalent. For example, Left Alt-y performs the same function as Right Alt-y.

You can choose to make the left Alt and right Alt keys work independently. That is, you can choose for the left Alt-y key sequence to perform a different function than the right Alt-y sequence. Similarly, you can choose for the left and right Shift keys to work independently. To make the left and right Alt keys or Shift keys work independently, click on the check box next to **ALT L = R** or **SHIFT L = R** in the bottom right of the configuration screen.

The left and right Amiga keys function independently. You cannot make these keys function equivalently.

- **Caution** *Do not define the Control, right Amiga key, left Amiga key sequence.* This key combination reboots the machine. ▲

To set or change the definition of a key or key sequence, follow these steps:

1. Select the key or key sequence by clicking on the appropriate keys. For example, to assign a command to the key sequence Control-C, press or click on the Control key and the C key.
2. Enter the command code in the text gadget. You can type in the command code, or you can select the code from the Keys pull-down menus. Press Return after entering the command code. For example, to assign the CW (Change Window) command to Control-C,

type in **CW** and press Return, or select **CW** from the Keys 2 menu and press Return.

You can repeat this step as many times as necessary to create complex key definitions.

3. Save the new settings as described in “Saving the New Settings,” later in this chapter.

Setting Tab Stops

By default, tab stops are set at every eighth character, as shown in Display 5.1.

You can clear all tab stops by clicking on the **Clear tabs** gadget. To set a tab stop, point and click to the place on the tab bar where you want to set a tab stop. An uppercase T indicates where a tab stop has been set. To delete a single tab stop, point and click at the tab stop.

After you have set at least two tab stops, you can set a series of tabs at equidistant intervals, using the **Repeat tabs** gadget. When you click on the **Repeat tabs** gadget, the configuration program calculates the interval between the last two tab stops you entered, and sets tab stops for the remainder of the tab bar using that interval.

Save the new tab stops as described in “Saving the New Settings,” later in this chapter.

Setting Other Options

The Options pull-down menu in the configuration menu allows you to specify several parameters, including how your input is processed, how the Search and Replace commands work, whether the editor saves backup files, and so on. The following list describes each of the options in the Options pull-down menu.

If you change any of the options in the Options pull-down menu, you must save your changes as described in “Saving the New Settings,” later in this chapter.

Input Processing

controls the mode in which you enter text into **se**. You can choose from five modes:

No processing does not convert anything to uppercase or insert carriage returns when you reach the right margin. You must press Return at the end of each line.

Asm modes are designed to make entering assembly language code easier. In the assembly language modes, the editor uses the tab stop settings to determine the start of the

next field. In Asm mode 1, the editor converts anything you enter in field 1 into uppercase. In Asm mode 2, the editor also converts anything you enter in field 2 into uppercase, and in Asm mode 3, it also converts anything you enter in field 3. The editor does not convert anything entered on a line after you enter a comment character (; or *).

In any of the assembly language modes:

- When you enter a colon (:), space, or tab character into the first field, the cursor moves to the second field.
- When you enter a space or tab character into the second field, the cursor moves to the third field. If you enter a comment character (; or *), the cursor moves to the comment field.
- When you enter a comment character into the third field, the cursor moves to the comment field.

You can enter a comment character into the text without causing the cursor to move to the comment field by preceding the comment character with a backslash (\).

Autowrap inserts carriage returns into the text when you reach the right margin. When the word you are entering extends past the right margin, the entire last word is moved to the next line.

The default setting is No processing. This option affects only the current window.

Tab Expansion Method

specifies whether tab characters are converted to spaces (Expand Tabs to Spaces) or left as tab characters (Use TAB Character) . Only the tab characters that are entered while Expand Tabs to Spaces is active are converted to spaces. Tab characters that already exist are not converted when you select Expand Tabs to Spaces. The default setting is Use TAB Character. This option affects only the current window.

Auto-Indent Mode

is designed to make entering C source code easier. When auto-indent mode is on, new lines are indented depending on the following criteria:

- If the previous line contains an open brace ({), the new line is indented one tab stop to the right of the previous line.

- ☐ Otherwise, if the character is a close brace (}), the new line is indented one tab stop to the left of the previous line.
- ☐ For any other character, the new line is indented to the same position as the previous line.

The default setting is On. This option affects only the current window.

Column Display

specifies whether the column number is displayed on the status line. The default setting is Off. This option affects all windows.

Prompt Before Undo

specifies whether you are asked to confirm each **undo** command. The default setting is On. This option affects all windows.

Backup File Mode

specifies how you want the editor to deal with backup files. You can specify one of the following:

- ☐ No Backup File indicates that you do not want the editor to save backup copies of each file that you edit.
- ☐ Place Backup File in Backup Dir specifies that you want the editor to save a backup copy of each file that you edit using the same filename and extension in a different directory. If you select this option, the configuration program asks you for the name of the backup directory.
- ☐ Rename Backup with Backup Ext specifies that you want the editor to save a backup copy of each file that you edit using the same filename in the same directory as the original file but with a different filename extension. If you select this option, the configuration program asks you to enter the backup file extension.

The default setting is No backup file. This option affects all windows.

Searches Method

specifies whether the Search and Replace commands should treat the search patterns you enter as simple character strings (String Matching) or as regular expressions (Use Regular Expressions). If you choose to Use Regular Expressions, you should enter search patterns as described in the description of the **grep** utility under “Specifying the Pattern” in *SAS/C Development System User’s Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0*. The default setting is Use Regular Expressions. This option affects all windows.

Case Sensitive Characters

specifies whether the Search and Replace commands are sensitive to case. For example, if this option is On and you specify a search pattern in lowercase, the Search command will not locate the same

pattern in the file if the pattern occurs only in uppercase. The default setting is On. This option affects all windows.

Saving the New Settings

To save any changes you make using the configuration window, select the Use, Save, or Save As options.

- ☐ Select **USE** to save the settings to the environment variable `sc/se.dat` in `ENV:`. Settings saved to `ENV:` are active until you reboot your system.
- ☐ Select **SAVE** to save the settings to `ENV:` and `ENVARC:` (the equivalent of `ENV:` on disk). Settings saved to `ENVARC:` are copied to `ENV:` each time you boot your system.
- ☐ Select the Save As option to save the settings in a file in the current directory. The configuration program asks you to enter a filename in which to save these settings.

If you click on the **Close** gadget without selecting the Use, Save, or Save As option, the new settings are active for your current `se` session only.

Reusing Saved Settings

If you use the Save As option to save your configuration settings, you can reload these settings later by selecting the Open option from the Project menu. The configuration program asks you to specify the filename to which you saved the settings. Enter the filename to which you saved the settings. To make the settings active, you must click on the **SAVE** or **USE** gadget.

7 Using the AREXX Interface to `se`

65	<i>Introduction</i>
65	<i>Executing A Series of Editor Commands</i>
68	<i>Communicating with External Programs</i>
69	<i>AREXX Macros Provided with the Compiler</i>
70	<i>AREXX Command Summary</i>

Introduction

This chapter describes the AREXX interface that is available in the `se` editor. AREXX is a script language commonly used for interprocess communication on Amiga hardware. AREXX comes with Version 2.0 of the AmigaDOS operating system and can be purchased separately from Commodore for Version 1.3. For more information about using AREXX, refer to *Using the System Software* (Commodore-Amiga, Inc. 1990) if you are running the AmigaDOS operating system, Version 2.0 or to the documentation that comes with the AREXX package if you are running AmigaDOS operating system, Version 1.3.

You can use the AREXX interface to do the following:

- ☐ execute a series of editor commands that are too complex to assign to a keystroke macro
- ☐ communicate with external programs.

Executing A Series of Editor Commands

Sometimes you need to perform an action that is more complex than `se` can handle with simple keystroke macros. For example, you may need to use programming logic such as an `if-then-else` construction. You can create an AREXX macro to perform the action, and then you can assign that macro to a key. AREXX macros are files that contain AREXX commands. AREXX macros that are to be used with `se` must have a filename with a `.se` suffix.

Tables 7.1 and 7.2 in “AREXX Command Summary,” later in this chapter describe each of the commands supported by AREXX. As you enter AREXX commands, separate each command with a space. You can string multiple commands together.

If you are passing characters to a command prompt, you must terminate the string with a newline character, `\n`. For example, the following command issues the Open Window command, enters `foo` for the filename, and terminates the string with a carriage return:

```
/* Issue the Open Window command,          */
/* type in "foo" for the filename,          */
/* and terminate data entry with a carriage return. */

'OW "foo\n"'
```

If you do not specify the `\n` character, `se` waits until you type a carriage return before proceeding.

After you have created your macro file, you can assign the macro to a key using the `se` configuration window. To display the `se` configuration window, select the Configuration Window option from the Options pull-down menu. Select the key to which you want to assign the macro by pressing or clicking on the appropriate key(s). Type the following Rexx Macro command in the text gadget and press Return:

```
RM "macro-filename"
```

The *macro-filename* is the name of the file containing your macro. If the name you specify does not end in `.se`, the editor appends `.se` to the filename when you invoke the macro. `se` searches for the macro file in the current directory, then in `sc:REXX`, and finally in `REXX:.` You can specify a fully-qualified pathname for the filename, if necessary.

For example, suppose you have a series of `#define` statements in your program such as the following:

```
#define FRED 1
#define BARNEY 4
#define WILMA 7
#define BETTY 8
```

You want to renumber these statements sequentially:

```
#define FRED 1
#define BARNEY 2
#define WILMA 3
#define BETTY 4
```

You could use the following AREXX macro. If invoked from `se`, this macro rennumbers the last token on each line of a marked block.

```

/* AREXX macro to renumber #define statements */

/* Tell SE to go to the Block Menu and select End. */
/* This effectively moves the cursor to the end of the */
/* current marked block. */
'BM "E"'

/* Ask SE for the line number of the current line. */
/* Assign the line number to the variable 'end'. */
options results
'GL'
end = result
options

/* Tell SE to go to the Block Menu and select Beginning. */
/* This moves the cursor to the beginning of the marked */
/* block. */
'BM "B"'

/* Ask SE for the text of the current line. */
/* Assign the text of the line to the variable 'line'. */
options results
'GT'
options
line = result

/* Move the cursor to the end of the current line. */
'EL'

/* Back up one word */
'PW'

/* Get the index of the current position in the current */
/* line. This index can differ from the current column */
/* number if tab characters are present in the line. */
/* Assign the index to the variable 'index'. */
options results
'GI'
options
index = result

```

```

/* Read the line and get the number at the end of it. */
/* Since the variable 'index' points to the first */
/* character of the number, the number consists of */
/* everything from the 'index'th byte of the line to the */
/* end of the line. */
num = SUBSTR(line, index+1, length(line)-index)

/* 'num' now contains the number at the end of the first */
/* line of the marked block. Move down a line, */
/* go to the end of the line, back up a word, and */
/* delete to the end of the line. The following */
/* sequence of four SE commands does this. */
'NL EL PW DE'

/* Loop until the end of the block, replacing the last */
/* token on the line with the appropriate number. */
do forever
  num = num + 1 /* Increment #define number */
  num /* Insert the number as text */
  options results /* Get the current line number */
  'GL'
  options
  if result >= end then exit(0) /* Exit if past the end */
  'NL EL PW DE' /* Go to next line and delete last token */
end

```

Communicating with External Programs

You can also use AREXX to communicate with other programs from within the editor. For example, you may want to send messages to the `scmsg` utility, which allows you to browse your error and warning messages. You can also invoke external commands that are unrelated to the editor or send AREXX commands to another program that has an AREXX port.

Each program that has an AREXX port names its port so that other programs can find it. The name of `se`'s AREXX port is `SC_SE`. If more than one copy of `se` is running, only one copy will have an AREXX port.

When you invoke an AREXX script from within an application, any external commands that the script attempts to execute are sent to the AREXX port of the application that launched it. Therefore, when `se` launches a script, it should send `se` commands as external commands. If the script wants to communicate with another utility, it should switch to that utility's AREXX port using the AREXX `ADDRESS` command. The

script can then send commands in whatever format required by that utility. To switch back to **se**, specify the **ADDRESS** command with **"SC_SE"** as its argument.

For example, the following **se** script tells **scmsg** to go to the next error, then gets the information on the error, and positions the cursor on the correct line number.

```

/* REXX script to advance to next error */
address "SC_SCMSG" /* Switch to the message browser. */
'NEXT'             /* Go to the next message on the list. */
options results    /* Remember the results of commands. */
'FILE'             /* Get the filename of the next error. */
file = result
'LINE'             /* Get the line number. */
line = result
options

address "SC_SE"     /* Back to the editor now */
'OW "||| file |||\n"' /* Issue Open Window command */
'LL "||| line |||\n"' /* Issue Go to line command */

```

AREXX Macros Provided with the Compiler

The SAS/C Development System provides seven AREXX macros that you can use from inside of **se**. The files for these macros are in **sc:rexx**. These macros are as follows:

amigaguide.se

displays the help information in the AmigaGuide databases in the **sc:help** directory. By default, this macro is assigned to Alt-D. When you invoke this macro, it asks you to enter the name of an AmigaGuide database. For a description of each of the AmigaGuide databases in the **sc:help** directory, see Chapter 3, "Getting Help."

deletetabs.se

converts all of the tabs within a marked block to spaces. By default, this macro is assigned to the asterisk (*) key in the numeric keypad (NUM *).

findsym.se

looks in the GSTs in memory for the definition of the symbol that is under the cursor. If it finds the symbol in a GST, **findsym** displays the source file in which the symbol is defined. If you do not have any GSTs in memory, this macro does nothing. By default, this macro is assigned to the left parenthesis key on the numeric keypad, NUM (.).

- indent.se**
indents a marked block three spaces. By default, this macro is assigned to the plus (+) key on the numeric keypad (NUM +).
- unindent.se**
unindents a marked block three spaces. By default, this macro is assigned to the minus (−) key on the numeric keypad (NUM −).
- tolower.se**
converts the current character to lowercase. This macro is not assigned to a key by default, but you can assign it to a key using the **se** Configuration Window.
- toupper.se**
converts the current character to uppercase. This macro is not assigned to a key by default, but you can assign it to a key using the **se** Configuration Window.

To assign any of these macros to a key, use the Rexx Macro command as described in “Executing A Series of Editor Commands,” earlier in this chapter.

AREXX Command Summary

The following tables list the AREXX commands and their keystroke equivalents supported by the editor.

Table 7.1
*AREXX Commands with
Keystroke Equivalents*

Command	Function	Default Key Definition
A1	assign macro 1	Alt-1
A2	assign macro 2	Alt-2
A3	assign macro 3	Alt-3
A4	assign macro 4	Alt-4
A5	assign macro 5	Alt-5
A6	assign macro 6	Alt-6
A7	assign macro 7	Alt-7
A8	assign macro 8	Alt-8
A9	assign macro 9	Alt-9

(continued)

Table 7.1
(continued)

Command	Function	Default Key Definition
A0	assign macro 0	Alt-0
BL	beginning of line	Num 7
BM	go to block menu	Ctrl-b
BT	beginning text	Ctrl-Num 7
CO	invoke scopts utility	Ctrl-F4
CS	change display size	F9
CW	change windows	Num 5
DC	delete character	Num .
DE	delete to end of line	Ctrl-e
DL	delete line	Ctrl-y
DW	delete word	Ctrl-w
EL	end of line	Num 1
ES	se escape character	Ctrl-\
ET	end text	Ctrl-Num 1
FB	toggle window front to back	F10
FD	go to fork command	Ctrl-f
FW	cycle through open windows	F6
GL	go to locate line command	Ctrl-l
HE	get se help	F1
IL	insert line	Ctrl-n
IM	toggle interlace mode	Enter
IN	toggle insert mode	Num 0
MB	mark beginning of block	Ctrl-[
MC	Match character	Ctrl-5
ME	mark end of block	Ctrl-]
MM	go to main se menu	F2

(continued)

Table 7.1
(continued)

Command	Function	Default Key Definition
M1	execute macro 1	Alt-F1
M2	execute macro 2	Alt-F2
M3	execute macro 3	Alt-F3
M4	execute macro 4	Alt-F4
M5	execute macro 5	Alt-F5
M6	execute macro 6	Alt-F6
M7	execute macro 7	Alt-F7
M8	execute macro 8	Alt-F8
M9	execute macro 9	Alt-F9
M0	execute macro 10	Alt-F10
NC	next character	Num 6
NE	go to next compiler error	F5
NL	next line	Num 2
NP	next page	Num 3
NW	next word	Ctrl-Num 6
OM	go to configuration menu	Ctrl-m
OW	go to open window command	Ctrl-o
PC	previous character	Num 4
PL	previous line	Num 8
PM	go to project menu	Ctrl-p
PP	previous page	Num 9
PW	previous word	Ctrl-Num 4
QU	go to quit se command	F3
RK	restore kill buffer	Alt-y
SA	go to next string match	Alt-s
SC	invoke SAS/C compiler	F4

(continued)

Table 7.1
(continued)

Command	Function	Default Key Definition
SD	scroll down	Ctrl-v
SE	go to search command	Ctrl-s
SR	go to search/replace command	Ctrl-r
SU	scroll up	Ctrl-6
UN	go to undo command	Ctrl-u

The following AREXX commands do not have keystroke equivalents.

Table 7.2
*AREXX Commands
without Keystroke
Equivalents*

Command	Function	Example
BP	build project	'BP'
CR	carriage control	'CR'
DM	display message	'DMExample Message'
GC	get column of cursor	'GC'
GE	get error from last command	'GE'
GI	get index of cursor	'GI'
GL	get number of current line	'GL'
GM	prompt user for input	'GM'
GT	get text of current line	'GT'
GW	get word	'GW'
RM	REXX macro	'RM "filename"'

■ Part 3

Using the Compiler and Linker

- Chapters**
- 8 Compiling and Linking Your Program
 - 9 Running Your Program from the Workbench™ Screen
 - 10 Using Startup Modules, Autoinitialization, and Autotermination Functions
 - 11 Using Amiga® Specific Features of the SAS/C® Language
 - 12 How Does the Compiler Work?
 - 13 Writing Portable Code

8 Compiling and Linking Your Program

77	<i>Introduction</i>
77	<i>Compiling Your Program</i>
77	<i>Entering the <code>sc</code> Command</i>
79	<i>Using Compiler Options</i>
119	<i>Using Preprocessor Symbols Defined by the Compiler</i>
120	<i>Linking Your Program</i>
121	<i>Entering The <code>slink</code> Command</i>
121	<i>Specifying Linker Options</i>
127	<i>Using Overlays</i>
132	<i>Specifying Compiler Options with Overlays</i>
133	<i>Customizing the Overlay Manager</i>

Introduction

This chapter describes how to compile and link your program. It describes the `sc` command and its options, and it describes the `slink` command and its options.

Compiling Your Program

As described in Chapter 2, “Using Your SAS/C Development System,” you can compile your program from the Shell by entering the `sc` command or from the Workbench screen by double-clicking on the **Build** icon. When you double-click the **Build** icon, you are indirectly running the `sc` command as well.

Entering the `sc` Command

The `sc` command has the following format:

```
sc [options] [source-filename(s)]
```

The `sc` command accepts C source files, assembly source files, object files, and link libraries as input files. You can enter the filenames anywhere on the `sc` command line, and you can use standard AmigaDOS wildcards to specify filenames.

sc determines the input file type by looking at the file extension. **sc** recognizes the following filename extensions:

- .c** C source file
- .p** C source file (preprocessor output)
- .h** C header file (treated as a C source file)
- .o** Object file
- .lib** Link library
- .a** Assembler source file
- .asm** Assembler source file
- .s** Assembler source file

If you have a file that does not end in one of the recognized extensions, you must use the correct **sc** option to specify the filename:

csource=filename

specifies a C source file. You can abbreviate this option as **csrc**.

assembler=filename

specifies an assembler source file. You can abbreviate this option as **asm**.

object=filename

specifies an object file. You can abbreviate this option as **obj**.

library=filename

specifies a link library. You can abbreviate this option as **lib**.

See “Compiler Option Descriptions,” later in this chapter for a complete description of each of these options.

If you specify an item in the **sc** command that does not end in a recognized extension and is not a valid option, **sc** appends the **.c** extension and attempts to compile the resultant filename. If the **sc** cannot find the file, it reports an error and proceeds to the next file to be compiled.

Note: You can use a plus (+) sign or a comma (,) to separate multiple filenames. Therefore, you cannot enter a filename containing a plus sign or a comma unless the name ends in a recognized extension.

To display help information about the **sc** command, enter **sc** followed by a question mark (?):

sc ?

Using Compiler Options

You can specify compiler options in the following ways:

- saving the options as project options in an **scoptions** file (using the **scopts** utility). The **sc** command looks for this file when you compile your program.
- saving the options as global default options in the variable **ENV:sc/scoptions** (using the **scopts** utility). If the **sc** command cannot find a project options file, it looks for global default options.
- entering the options in the **sc** command. Even if you are using an **scoptions** file, you can also specify options on the **sc** command line.

When you enter the **sc** command, the compiler first tries to read options from the file **scoptions** in the current directory. If this file does not exist, the compiler looks for options stored in the **ENV:sc/scoptions** environment variable. Finally, the compiler processes any options you enter on the **sc** command line in the order in which you enter them. If an option is specified twice, the compiler uses the setting of that option that was specified last. Therefore, you can override the setting of an option in the **scoptions** file or environment variable by specifying that option in the **sc** command. **sc** reads all options before processing any files, so any options specified on the command line after an input file name still apply to that file.

If your current directory contains an **scoptions** file but you do not want the compiler to use the options specified in the file, specify the **resetoptions** option as the first option in the **sc** command. **resetoptions** sets all options to their default values. **resetoptions** does not change the contents of the **scoptions** file but does affect the compilation in progress. For more information, see the description of the **resetoptions** option later in this section.

The compiler accepts two types of options:

switch options

do not take parameters. All switches have a negative and a positive version. To specify the negative version, precede the option with **no**. For example, to use ANSI escape sequences to highlight errors in diagnostic output, specify the **errorhighlight** option. If you do not want to highlight errors, specify **noerrorhighlight**.

keyword options

take parameters. Specify the option name, followed by a blank or an equal (=) sign and the parameter. For example, to generate debugging information for your program, you can specify **debug=line** or **debug line**. Some keyword options also have a negative form like the switch options. For example, if you do not want the compiler to generate debugging information, you specify **nodebug**.

The option descriptions that follow give the full name and minimum acceptable abbreviation for each option (if the option has an abbreviation). You can also use intermediate forms of the options. For example, the minimum abbreviation for the **absfunctionpointer** option is **afp**. Other acceptable abbreviations include **absfp**, **absfuncp**, and **afptr**. The compiler options are not case-sensitive. You can enter options in upper, lower, or mixed case.

Summary of Compiler Options

For each option accepted by the compiler, Table 8.1 shows:

- the full name.
- the minimum abbreviation. The minimum acceptable abbreviation is shown with uppercase letters.
- whether the option takes a parameter. An option that takes a parameter is listed with an equals (=) sign.
- whether the option has a negative form.
- the default value, if any.

Table 8.1
Summary of Compiler Options

Option Name	Negative Form?	Default Value
AbsFuncPointer	Yes	noafp
ADDSYMBOLS	Yes	noaddsym
ANSI	Yes	noansi
ARGUMENTSIZE=	No	512
ASSEMBLER=	No	
AUTOREGISTER	Yes	autoreg
BATCH	Yes	nobatch
BSSMEMORY=	No	any
BSSNAME=	Yes	udata
CODE=	No	near
CODEMEMORY=	No	any
CODENAME=	Yes	text
CommentNEST	Yes	nocnest
COMMON	Yes	nocommon

(continued)

Table 8.1
(continued)

Option Name	Negative Form?	Default Value
CONSTLIBbase	Yes	constlib
COVERage	Yes	nocover
CPU=	No	any
CSouRCe=	No	
DATA=	No	near
DATAMemory=	No	any
DATANAME=	Yes	data
DeBuG=	Yes	nodebug
DEFine=	No	
DISASsemble=	Yes	nodisasm
ERRor=	No	
ERRorCONsole	Yes	errcon
ERRorHIGHlight	Yes	errhigh
ERRorLIST	Yes	errlist
ERRorREXX	Yes	errrexx
ERRorSouRCe	Yes	errsrc
EXTErnalDEFs	Yes	extdef
FindSYMBOL=	No	
FROM	No	
GenProtoDATAitems	Yes	gpdata
GenProtoEXTerns	Yes	gpext
GenProtoFILE=	Yes	filename_protos.h
GenProtoPARAmeters	Yes	nogpparm
GenPROTOS	Yes	nogproto
GenProtoSTATics	Yes	nogpstat

(continued)

Table 8.1
(continued)

Option Name	Negative Form?	Default Value
GenProtoTypeDEFS	Yes	gptdef
GlobalSymbolTable=	Yes	nogst
GSTIMMediate	Yes	nogstimm
ICONS	Yes	icons
IDentifierLENgth=	No	255
IGNore=	No	
IncludeDIRectory=	No	
KeepLINE	Yes	nokline
LIBCODE	Yes	nolibcode
LIBrary=	No	
LINK	Yes	nolink
LINKerOPTions=	Yes	nolinkopt
LIST	Yes	nolist
ListFILE=	Yes	filename.lst
ListHEADers	Yes	nolhead
ListINCludes	Yes	listinc
ListMACros	Yes	nolmac
ListNARRow	Yes	lnarr
ListSYStem	Yes	nolsys
MakeGlobalSymbolTable=	Yes	
MAP	Yes	nomap
MapFILE=	Yes	executable.map
MapHUNK	Yes	nomaphunk
MapLIB	Yes	nomlib
MapOverLaY	Yes	nomovly
MapSYMBOLs	Yes	nomsym
MapXREference	Yes	nomxref

(continued)

Table 8.1
(continued)

Option Name	Negative Form?	Default Value
MATH=	Yes	nomath
MAXimumERRORs=	Yes	50
MAXimumWARNings=	Yes	nomaxwrn
MEMorySIZE=	No	large
MODified	Yes	nomod
MultipleCharacterCONSTants	Yes	nomccons
MultipleINCludes	Yes	minc
OBJect=	No	
OBJectLIBrary=	Yes	
OBJectNAME=	Yes	
OLDPreProcessor	Yes	nooldpp
ONERROR=	No	stop
OPTimize	Yes	noopt
OPTimizerALIAS	Yes	optalias
OPTimizerCOMplexity=	Yes	3
OPTimizerDEPth=	Yes	3
OPTimizerGloBal	Yes	optgo
OPTimizerINLine	Yes	optinl
OPTimizerINLOCAL	Yes	nooptinlocal
OPTimizerLOOP	Yes	optloop
OPTimizerPEEPhole	Yes	optpeep
OPTimizerRecurDEPth=	Yes	3
OPTimizerSIZE	Yes	nooptsize
OPTimizerTIME	Yes	noopttime
PARaMeters=	No	stack
PRECision=	No	mixed
PreProcessorBUFFer=	No	8192
PreProcessorONLY	Yes	nopponly
ProgramNAME=	No	

(continued)

Table 8.1
(continued)

Option Name	Negative Form?	Default Value
RESetOPTions	No	
SAVEDS	Yes	nosaveds
ShortINTegers	Yes	nosint
SmallCODE	Yes	noscode
SmallDATA	Yes	nosdata
SourceIS=	Yes	
STacKHeck	Yes	stkchk
STacKEXtend	Yes	nostkext
STanDardIO	Yes	stdio
STaRTup=	Yes	c (for c.o)
STRICT	Yes	nostrict
STRingsCONst	Yes	nostrcons
STRingMERge	Yes	nostrmer
STRIPDeBuG	Yes	nostripdbg
STRuctureEQuivalence	Yes	nostreq
TO=	No	
TRIGraph	Yes	notrig
UnderSCORE	Yes	nouscore
UnsignedCHAR	Yes	nouchar
UTILityLIBrary	Yes	noutillib
VERBOSE	Yes	noverbose
VERSION	Yes	version
WaRN=	No	
WarnVoidRETurn	Yes	wvret
WITH=	No	
XREference	Yes	noxref
XreferenceHEADers	Yes	noxhead
XreferenceSYStem	Yes	noxsys

Compiler Option Descriptions

The following pages describe each of the options accepted by the `sc` command. The compiler options are not case-sensitive. You can enter options in upper, lower, or mixed case.

AbsFuncPointer

generates 32-bit references to functions when loading function pointers. The default value is `noabsfuncpointer`. The minimum acceptable abbreviation is `afp`.

If you specify `noabsfuncpointer` and you take the address of a function that is more than 32k away, the linker generates an `ALV` (automatic link vector) jump instruction that allows your code to work. However, if you compare the address of the function to a function pointer assigned elsewhere, you may get a different value. Specify `absfuncpointer` if your code is larger than 64K or in multiple code hunks and you compare function pointers. Specifying `absfuncpointer` may increase the size of your executable.

AddSymbols

tells the linker to add symbol information to the executable module. The default value is `noaddsymbols`. The minimum acceptable abbreviation is `addsym`.

This option is automatically enabled if you specify the `debug` option. This option is ignored if you do not specify the `link` option.

ANSI

enforces the strictest interpretation of the ANSI C standard. Using this option produces many additional warning messages. The default value is `noansi`.

To be completely ANSI-compliant (that is, if you require a pure ANSI namespace), you should also define the preprocessor symbol `__STRICT_ANSI` before including any header files. For more information about `__STRICT_ANSI`, refer to Chapter 7, "Library Reference," in *SAS/C Development System Library Reference, Version 6.0*. If you require support for ANSI trigraphs, specify the `trigraph` option also.

For more information about improving the portability of your code, see Chapter 13, "Writing Portable Code."

ArgumentSize=n

sets the size of the maximum argument to a preprocessor macro. The minimum acceptable abbreviation is `argsiz`. This option does not have a negative form. The default value is `512`. However, the

memorysize option sets the **argumentsize** (and other internal limits) to different default values if you do not specify **argumentsize**.

See the description of the **memorysize** option for more information.

Assembler=filename(s)

specifies assembly-language files that are to be assembled and, if you specify the **link** option, linked into the program. The minimum acceptable abbreviation is **asm**. This option does not have a negative form.

You can use AmigaDOS wildcard characters to specify filenames. To specify several filenames or wildcard patterns, separate each filename with a plus (+) sign or a comma (.). You can specify the **assembler** option as many times as necessary.

If you are assembling a disassembly that was generated with the **disassemble** compiler option, specify the **underscore** compiler option also. (If you assemble a disassembly by calling the assembler directly, specify the **-u** assembler option.)

See also the descriptions of the **csource**, **object**, and **library** options.

AutoRegister

enables automatic register selection by the code generator. The default value is **autoregister**. The minimum acceptable abbreviation is **autoreg**.

If you specify **autoregister**, the compiler attempts to add register variables to the variables that have already been chosen by the global optimizer or declared with the **register** keyword.

Batch

tells the linker not to prompt for definitions of undefined symbols. The default value is **nobatch**. This option is ignored if you do not specify the **link** option. See also the description of the **batch** linker option.

BSSMemory=type

specifies the type of memory into which uninitialized external data items should be loaded. You can specify **any**, **chip**, or **fast** for **type**. You can abbreviate these values as **a**, **c**, or **f**. The default value is **any**. This option does not have a negative form.

This option affects code generated by both the compiler and the assembler. See also the descriptions of the **datamem** and **codemem** options.

BSSName=name

names the uninitialized data section. The default value is **udata**. You can specify **bssname=none** or **nobssname** if you want an unnamed BSS section. The linker automatically merges all sections with the same name. See also the descriptions of the **codename** and **dataname** options. See Chapter 12, “How Does the Compiler Work?” for information on data and code sections.

Code=reference-type

specifies whether you want 16-bit or 32-bit references to functions not declared in the current file. You can specify **near** or **n** for 16-bit references or **far** or **f** for 32-bit references. The default value is **near**. This option does not have a negative form.

Most programs do not need this option even if they are very large, because the linker creates a jump instruction (an **ALV**) for any references to functions that are out of range. See also the description of the **data** option.

CodeMemory=type

specifies the type of memory into which code should be loaded. You can specify **any**, **chip**, or **fast**. You can abbreviate these values as **a**, **c**, or **f**. The default value is **any**. This option does not have a negative form.

This option affects code generated by both the compiler and the assembler. See also the descriptions of the **bssmem** and **datamem** options.

CodeName=name

names the code section. The default value is **text**. You can specify **codename=none** or **nocodename** if you want an unnamed code section. The linker automatically merges all sections with the same name. See also the descriptions of the **bssname** and **dataname** options. See Chapter 12, “How Does the Compiler Work?” for information on data and code sections.

CommentNest

allows nested comments. The default value is **nocommentnest**. The minimum acceptable abbreviation is **cnest**.

Nested comments occur when one comment is totally contained inside another. The ANSI Standard prohibits nested comments, so in ANSI-compliant code, the first comment end sequence (***/**) terminates both comments. For example, the statement below generates an error

with **nocommentnest**, but compiles successfully with **commentnest**.

```
/* i = i+1; /* This is a comment */ */
```

The statement below compiles successfully with **nocommentnest**, but generates an error with **commentnest**.

```
/* i = i+1; /* This is a comment */
```

Common

tells the compiler to use the common model for external data. If you specify **nocommon**, the compiler uses the strict reference-definition model. The default value is **nocommon**.

The section “Using Common Model External Data” in Chapter 11, “Using Amiga Specific Features of the SAS/C Language,” describes common and strict reference-definition models.

ConstLibBase

tells the compiler that library base pointers are set once then remain constant throughout your entire program. The default value is **constlibbase**. The minimum acceptable abbreviation is **constlib**.

If you change the value of your library bases after setting them, you should specify the **noconstlibbase** option. Specifying **constlibbase** allows the compiler to prevent extra register loading when making a series of calls to library functions through an external variable containing the library base.

Coverage

tells the compiler to generate code to collect coverage analysis information. The default value is **nocover**. The minimum acceptable abbreviation is **cover**.

Coverage analysis information allows you to determine which lines of your program have been executed by your test cases. For more information, refer to the description of the **cover** utility in *SAS/C Development System User’s Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0*.

If you specify the **cover** option, make sure that you:

- ❑ Link with the file **sc:examples/cover/covutil.o** or **LIB:covutil.o**.
- ❑ Do not specify the **nostdio** option.
- ❑ Use **startup=c** to link with the standard **c.o** startup module.

CPU=processor

generates code specific to the specified processor. You can specify **any**, **a**, or **68000** to generate code for any processor. You can also specify **68010**, **68020**, **68030**, or **68040** to generate code for a specific processor. The default value is **any**. This option does not have a negative form.

This option defines one or more preprocessor symbols. See the section “Using Preprocessor Symbols Defined by the Compiler,” later in this chapter for a list of those symbols.

This option affects code generated by both the compiler and the assembler.

CSource=filename

specifies C source files that are to be compiled and, if you specify the **link** option, linked into the program. The minimum acceptable abbreviation is **csrc**. This option does not have a negative form.

You can use AmigaDOS wildcard characters to specify filenames. To specify several filenames or wildcard patterns, separate each filename with a plus (+) sign or a comma (.). You can specify the **csource** option as many times as necessary. See also the descriptions of the **assembler**, **object**, and **library** options.

Data=reference-type

specifies whether you want the compiler to generate 16-bit or 32-bit references to external and static data items. You can specify any of the following:

near or n

tells the compiler to use 16-bit references. If you specify **near**, all data not declared with the **__far** or **__chip** keyword are placed into the near data section. The default value is **near**.

far or f

tells the compiler to use 32-bit references. Register A4 is still reserved to point to the near data section so that you can mix code compiled with **data=near** and **data=far**.

faronly or fo

tells the compiler that your program never uses near data. If you specify **faronly**, the compiler generates 32-bit references and may use register A4 as an additional register variable. If you compile with the **data=faronly** option, and you declare data with the **__near** keyword, the compiler displays the warning message 194:

```
too much local data for NEAR reference,
some changed to FAR
```

If your entire project is in one source file or is compiled with **data=faronly**, you can ignore this warning unless you get an error later in the compilation or link.

auto or a

indicates that the first 64k of external data should generate 16-bit references and the remaining external data should generate 32-bit references. This option does not have a negative form.

You can override this option on individual data items by using the **__near**, **__far**, or **__chip** keywords. **__near** forces the compiler to generate a 16-bit reference, and **__far** forces the compiler to generate a 32-bit reference. **__chip** forces the compiler to place the data item into chip memory. For more information, refer to the section “Using Special Keywords” in Chapter 11, “Using Amiga Specific Features of the SAS/C Language.”

See also the description of the **code** option.

DataMemory=type

specifies the type of memory into which initialized static or external data should be loaded. You can specify **any**, **chip**, or **fast**. You can abbreviate these values as **a**, **c**, or **f**. The default value is **any**. This option does not have a negative form.

This option affects code generated by both the compiler and the assembler. See also the descriptions of the **bssmem** and **codemem** options.

DataName=name

names the initialized data section. The default value is **data**. You can specify **dataname=none** or **nodataname** if you want an unnamed data section.

The linker automatically merges all sections with the same name. See also the descriptions of the **bssname** and **codename** options. See Chapter 12, “How Does the Compiler Work?” for information on data and code sections.

Debug=level

sets the debugging level of the compiler. If you do not want the compiler to generate debugging information, specify **nodebug**. The default value is **nodebug**. The minimum acceptable abbreviation is **dbg**.

To generate debugging information, specify **debug=level**, where *level* is one of the following:

line or **l**

produces line number information only.

symbol or **s**

produces line number information, information on automatic and formal variables, and information on external and static symbols that are referenced in the module being compiled.

symbolflush or **sf**

produces the same information as **symbol**, and flushes any non-register variables being held in registers to memory at each line boundary to allow the debugger to accurately display their values in C source mode.

full or **f**

produces the same information as **symbol**. However, **debug=full** produces information on all symbols whether or not the module references the symbol.

fullflush or **ff**

produces the same information as **full**, and flushes any non-register variables being held in registers to memory at each line boundary to allow the debugger to accurately display their values in C source mode.

Any debug option except **nodebug** adds the **-d** assembler option to any assembled files to force debugging line number information on assembler output. Also, if you specify the **link** option and any debug option except **nodebug**, the **addsym** option is passed to the linker.

Define[=]symbol[=value]

defines the specified preprocessor symbol, as if with a **#define** statement. The minimum acceptable abbreviation is **def**. This option does not have a negative form.

Do not enter a space between the symbol name and the following equal sign. If the value contains a space, enclose the entire argument in double quotes (""). As with all other compiler options, the equal sign between the **define** option and the symbol name is optional, so both of the following examples are acceptable:

```
define foo=bar
define=foo=bar
```

You can specify the **define** option as many times as necessary.

Any symbols defined with the **define** option are defined in the assembler as well.

Note: The **define** compiler option does not affect the linker. Do not confuse this option with the **define** linker option.

DisAssemble=filename

tells the compiler to disassemble the code as it is generated and to send the disassembly to the file you specify. The default value is **nodisassemble**. The minimum acceptable abbreviation is **disasm**.

To send the disassembly to the Shell, use **disasm=***.

Error=n

tells the compiler to treat the specified message as an error. You can specify **all** or **a** to promote all enabled warnings to errors, or you can specify one or more message numbers to promote only those messages. The minimum acceptable abbreviation is **err**. This option does not have a negative form.

To specify several message numbers, separate each number with a plus (+) sign or a comma (,). You can specify the **error** option as many times as necessary. See also the descriptions of the **warn** and **ignore** options.

ErrorConsole

enables printing of diagnostics to the console (**stdout**). The default value is **errconsole**. The minimum acceptable abbreviation is **errcon**.

ErrorHighlight

highlights the token that caused the error using ANSI escape sequences in diagnostic output that is sent to the console. The default value is **errorhighlight**. The minimum acceptable abbreviation is **errhigh**.

ErrorList

prints diagnostic messages to the listing file. The default value is **errorlist**. The minimum acceptable abbreviation is **errlist**. This option is ignored if you do not specify the **list** option.

ErrorRexx

sends diagnostic messages to the **scmsg** utility. The default value is **noerrorrex**. The minimum acceptable abbreviation is **errrex**.

For more information, refer to the description of the **scmsg** utility in *SAS/C Development System User's Guide, Volume 2*.

ErrorSource

prints lines from the C source file with the diagnostic messages that are sent to the console. The default value is **errorsource**. The minimum acceptable abbreviation is **errsrc**.

ExternalDefs

treats all external definitions as definitions. The default value is **externaldefs**. The minimum acceptable abbreviation is **extdef**.

If you specify **noexternaldefs**, all external definitions are treated as external declarations. This action has the same effect as if you had declared each variable with the **extern** keyword. For example, **int i** is treated as **extern int i**, and it would need to be defined in a file compiled without **noexternaldefs**.

FindSymbol=*symbol-name*

tells the compiler to print a warning message each time the specified symbol is defined. The minimum abbreviation is **fsym**. This option does not have a negative form. You can specify the **findsymbol** option as many times as necessary.

If you specify the **findsymbol** option, a message is produced for any definition of the symbol, including **#define** statements, structure and union declarations, prototypes, and **extern**, **static**, and **local** variable definitions. Use the **fsym** option to quickly determine where a given preprocessor symbol or prototype is coming from.

From

is included only for compatibility with the **slink** command. The compiler ignores this option. This option does not have a negative form.

GenProtoDataItems

generates external declarations for variables defined in the source files that are not defined as **static**. The minimum abbreviation is **gpdata**. The default value is **gpdata**. This option is ignored if you do not specify the **genprotos** option.

GenProtoExterns

generates prototypes for externally-known routines. The default value is **genprotoexterns**. The minimum acceptable abbreviation is **gpext**.

This option is ignored if you do not specify the **genprotos** option.

GenProtoFile=filename

specifies the name of the file in which to place the generated prototypes. The default value is **filename_protos.h**. The minimum acceptable abbreviation is **gpfile**.

This option is ignored if you do not specify the **genprotos** option.

GenProtoParameters

generates prototypes using the **__PARMS** macro. The default value is **nogenprotoparameters**. The minimum acceptable abbreviation is **gpparm**.

This option allows your C code to compile successfully on compilers that support prototypes and on those that do not. On ANSI compilers, the **__PARMS** macro expands to the parameter list for the function, thereby creating a prototype. On non-ANSI compilers, the **__PARMS** macro expands to an open-close parentheses pair, which declares the function's return type without defining a prototype. This option is ignored if you do not specify the **genprotos** option.

GenProtos

generates prototypes and data declarations instead of compiling your file. The default value is **nogenprotos**. The minimum acceptable abbreviation is **gproto**.

This option defines the preprocessor symbol **_GENPROTO**. If you specify a filename with the **genprotofile** option, the prototypes are

written to the specified file. Otherwise, the prototypes are written to the file `filename_protos.h`.

While generating prototypes, the compiler suppresses most warnings automatically, because many of the warnings may be due to incorrect or missing prototypes. The compiler also checks all `#include` statements as they are reached. If your file `#includes` the same prototype file that is being generated, the compiler skips that `#include` statement. This feature allows you to use this option to maintain declarations for all externally-known symbols in each C source file and regenerate the declarations as the files change.

To set up your project so that you can use this option to maintain prototype files, do the following:

1. Create a header file that contains `#include` statements for each of the files in your project, as follows:

```
#include "file1_protos.h"
#include "file2_protos.h"
.
.
.
#include "filen_protos.h"
```

2. Include this header file in each file in your project.
3. Compile your entire project with the `genproto` option.

As each `.c` file is compiled, the compiler creates the corresponding `_protos.h` file. The compiler suppresses the `header file not found` warnings that would normally be produced.

GenProtoStatics

generates prototypes for static routines. The default value is `nogenprotostatics`. The minimum acceptable abbreviation is `gpstat`.

This option is ignored if you do not specify the `genprotos` option.

GenProtoTypedefs

tells the compiler to use typedefs instead of resolved types when generating prototypes for any functions using typedefs for parameters or return values. The default value is `genprototypedefs`. The minimum acceptable abbreviation is `gptdef`.

This option is ignored if you do not specify the `genprotos` option.

GlobalSymbolTable=gst

tells the compiler to use the specified GST (Global Symbol Table). The default value is **noglobalsymboltable**. The minimum acceptable abbreviation is **gst**.

The GST must have been created using the **makeglobalsymboltable** option during a previous compilation. The **gst** option is ignored if you specify the **makeglobalsymboltable** option. Therefore, you can enter the **makeglobalsymboltable** in the **sc** command even if your **scoptions** file contains the **gst** option.

This option defines the preprocessor symbol **_GST**.

GST=gst-filename

is a synonym for the **GlobalSymbolTable** option.

GSTImmediate

is included for compatibility with projects using precompiled header files as implemented in Version 5 of the compiler. This option makes the contents of the GST you specify with the **gst** option immediately available to the program. The default value is **nogstimmmediate**. The minimum acceptable abbreviation is **gstimm**.

Normally, symbols defined in a specific header file in the GST are available to your program only after you have included the header file with a **#include** statement. This option makes the contents of the GST you specify with the **gst** option immediately available to the program, even if your program does not contain **#include** statements for the header file. With Version 5 precompiled header files, all symbols in the precompiled header files were available to your program even if your program did not contain **#include** statements for the header file.

Icons

tells the compiler to create icons for files that it creates, including listing files, preprocessor output files, prototype files, and object files. The default value is **icon**.

If you specify **icons**, then each time the compiler generates a file, it looks in the drawer **sc:icons** for an icon named **def_extension**, where **extension** is the filename extension of the file it created. If the compiler finds an icon file appropriate to the file extension, it copies the icon to the directory in which the file was created. If the compiler cannot find **sc:icons** or cannot find an icon with the appropriate extension, it does not create an icon. If you

specify **noicons** and **link**, the **noicons** option is also passed to the linker. For more information, see the section “Using Icons” in Chapter 2, “Using Your SAS/C Development System.”

IdentifierLength=*n*

specifies the maximum number of significant characters in an identifier. The default value is 255. The minimum acceptable abbreviation is **idlen**. This option does not have a negative form.

Identifiers longer than *n* are truncated without warning. Identifiers longer than *n* bytes that differ after the first *n* bytes are treated as identical.

Ignore=*n*

tells the compiler to ignore the specified warning message. The minimum acceptable abbreviation is **ign**. This option does not have a negative form.

You cannot ignore error messages. You can specify **all** or **a** to ignore all warning messages, or you can specify one or more message numbers to ignore only those messages. To specify several message numbers, separate each number with a plus (+) sign or a comma (,). You can specify the **ignore** option as many times as necessary.

See also the descriptions of the **error** and **warn** options.

IncludeDirectory=*directory*

adds a directory to the list of directories to search for include files. The default list is the current directory and **include:**. The minimum acceptable abbreviation is **idir**. This option does not have a negative form. You can specify the **incdirectory** option as many times as necessary.

Any directories you specify with **includedirectory** are also passed to the assembler as header file search directories.

KeepLine

generates **#line** directives in the preprocessor output file that correspond to the lines in the original source files. The default value is **nokeepline**. The minimum abbreviation is **kline**.

This option allows you to compile the preprocessed source and get error and warning messages that refer you to the correct line in the nonpreprocessed version of the file.

This option is ignored if you do not specify the **pponly** option.

LibCode

tells the compiler that the compiled code will be linked into a shared library. The default value is **nolibcode**.

Any functions compiled with **libcode** and either the **__saveds** keyword or the **saveds** option load their near data section from a point relative to the library base register A6 instead of from a global data area. **libcode** also guarantees that the current library base will be in register A6 whenever A6 is referenced or an internal call is made.

Do not use this option when creating a normal executable module.

Library=link-library-filename(s)

specifies the link libraries that are to be passed to the linker. The minimum acceptable abbreviation is **lib**. This option does not have a negative form.

You can use AmigaDOS wildcard characters to specify filenames. To specify several filenames or wildcard patterns, separate each filename with a plus (+) sign or a comma (,). You can specify the **library** option as many times as necessary. Any libraries you specify are passed to the linker before the SAS/C libraries. This option is ignored if you do not specify the **link** option.

See also the description of the **csource**, **object**, and **asm** options.

Link

tells the compiler to invoke the linker to produce a final executable. The default value is **nolink**.

If you do not specify **link**, the compiler ignores any object files and link libraries that you specify.

The options passed to the linker are placed into the file **program.lnk**, and the linker is invoked using this file as a **with** file. To see which linker options were generated, look at the **.lnk** file after **sc** runs the linker.

LinkerOptions=option(s)

passes the provided parameter to the linker as command line options. The default value is **nolinkeroptions**. The minimum acceptable abbreviation is **linkopt**.

If you want to specify more than one option, or if the option you want to specify contains a blank, surround the entire option string with double quotes ("), as in the following example:

```
sc linkeroptions="bufsize 10000 maxhunk 2" link myprog.c
```

Any options specified in the options string are passed to the linker after any compiler options that are identified as valid only if you specify **link**. Therefore, the **linkeroptions** values override the values passed by the **sc** options.

This option is ignored if you do not specify the **link** option.

List

tells the compiler or assembler to produce a listing file. The default value is **nolist**.

The compiler writes the output to the filename you specify with the **listfile** option. If you do not specify an output filename, the compiler uses the name of the first source file you specify but with the extension **.lst**.

ListFile=filename

names the listing and/or cross reference file. The default filename is the same filename as the source file but with the extension **.lst**. The minimum acceptable abbreviation is **lfile**.

This option is ignored if you do not specify the **list** or **xreference** options.

ListHeaders

tells the compiler or assembler to include user header files in the listing. The default value is **listheaders**. The minimum acceptable abbreviation is **lhead**.

This option is ignored if you do not specify the **list** option.

ListIncludes

lists the names of all included **.h** files in the listing file. The default value is **listinc**. The minimum acceptable abbreviation is **linc**.

This option is useful for determining:

- ☐ **.h** file dependencies for **smake**
- ☐ the exact path of each **.h** file used
- ☐ exactly which **.h** files are included by other **.h** files.

The hierarchy of files is indicated by indentation levels; a file included by another file is indented one level deeper than the parent file. This option is ignored if you do not specify the **list** option.

ListMacros

tells the compiler or assembler to expand macros in the listing. The default value is **nolistmacros**. The minimum acceptable abbreviation is **lmac**.

This option is ignored if you do not specify the **list** option.

ListNarrow

tells the compiler or assembler to produce a narrow listing (less than 80 columns wide). The default value is **listnarrow**. The minimum acceptable abbreviation is **lnarr**.

This option is ignored if you do not specify the **list** option.

ListSystem

tells the compiler to include system header files in the listing. The default value is **nolistsystem**. The minimum acceptable abbreviation is **lsys**.

This option is ignored if you do not specify the **list** option.

MakeGlobalSymbolTable=gst-filename

creates a GST (Global Symbol Table). The minimum acceptable abbreviation is **mgst**.

If you specify the **gst** and **makegst** options, the **gst** option is ignored. Therefore, you can enter the **makegst** in the **sc** command even if your **scoptions** file contains the **gst** option. This option automatically enables the **nomultipleincludes** and **noexternaldefs** options. For more information on creating and using GSTs, refer to *SAS/C Development System Library Reference*.

This option defines the preprocessor symbol **_MGST**.

Map

produces a map of the executable module. The default value is **nomap**. This option is ignored if you do not specify the **link** option.

MapFile=filename

names the map file. The default value is **executable.map**. The minimum acceptable abbreviation is **mfile**.

This option is ignored if you do not specify the **link** and **map** options.

MapHunk

maps all output hunks by size and originating function. The default value is **nomaphunk**. The minimum acceptable abbreviation is **nomhunk**.

This option is ignored if you do not specify the **link** and **map** options.

MapLib

generates a list of hunks by library symbol in the map. The default value is **nomaplib**. The minimum acceptable abbreviation is **mlib**.

This option is ignored if you do not specify the **link** and **map** options.

MapOverlay

includes a list of hunks in each overlay in the map. The default value is **nomapoverlay**. The minimum acceptable abbreviation is **movly**.

This option is ignored if you do not specify the **link** and **map** options.

MapSymbols

includes a list of defined symbols and the location at which they are defined. The default value is **nomapsymbols**. The minimum acceptable abbreviation is **msym**.

This option is ignored if you do not specify the **link** and **map** options.

MapXreference

writes a symbol cross-reference to the map file that lists each symbol definition and the places each symbol is used. The default value is **nomapxreference**. The minimum acceptable abbreviation is **mxref**.

This option can generate a lot of output, but it is useful when you are trying to track down where an unresolved symbol is referenced. This option is ignored if you do not specify the **link** and **map** options.

Math=type

chooses a format for floating-point math and, if you also specify the **link** option, links with the appropriate math library. The default value is **nomath**.

You can specify one of the following as the **type**:

standard or s

links with the library **scm.lib** (or with **scms.lib** if you specify the **shortint** option). The math format is IEEE.

ffp or f

generates code to call the FFP shared library provided by Commodore. The math format is FFP.

68881, 68882, or 8

generates inline code for the 68881 and 68882 coprocessors. If you specify one of these coprocessor options, your program will not run if a coprocessor is not available. **coprocessor** is a synonym for these options. The math format is IEEE.

ieee or i

generates code to call the IEEE shared library provided by Commodore. The math format is IEEE.

Some math options define preprocessor symbols. See the section “Using Preprocessor Symbols Defined by the Compiler,” later in this chapter, for a list of those symbols.

For additional information about compiling and linking with math libraries, refer to Chapter 3, “Using the SAS/C Libraries,” in *SAS/C Development System Library Reference*.

MaximumErrors=n

sets the limit on the number of errors for a single compilation. The default value is **50**. The minimum acceptable abbreviation is **maxerr**.

If a single compilation generates more than *n* errors, the compiler aborts the compilation. **nomaxerr** removes any limits.

MaximumWarnings=n

sets the limit on the number of warnings for a single compilation. The default value is **nomaxwrn**. The minimum acceptable abbreviation is **maxwrn**.

If a single compilation generates more than *n* warnings, the compiler aborts the compilation. **nomaxwrn** removes any limits.

MemorySize=size

tells the compiler approximately how much memory you have on your system. You can specify one of the following:

- ☐ **tiny** or **t**
- ☐ **small** or **s**
- ☐ **medium** or **m**
- ☐ **large** or **l**
- ☐ **huge** or **h**

The default value is **large**. The minimum acceptable abbreviation is **memsize**. This option does not have a negative form.

Larger sizes allow **sc** to compile more complex programs and to compile faster. Smaller sizes allow **sc** to continue to work under low-memory conditions. If the compiler runs out of memory during a compilation, it displays the message *****Freeing Resources**, attempts to free up memory, and automatically drops to a lower **memorysize** value.

memorysize affects how and where the compiler stores any debugging information. If the compiler begins to run out of memory, it starts writing debugging information to a disk file. This file is referred to as a *debug side file*.

memorysize also affects the disposition and buffering of the compiler intermediate file. This option tells the compiler how much initial memory space to reserve for the intermediate information. For **large** and **huge**, the intermediate file is kept totally in memory, which is much faster than writing it to disk when the memory is available. At smaller values, the intermediate file is written to disk, but the **memorysize** value affects the amount that is buffered.

In addition, if you do not specify the **preprocessorbuffer** (**ppbuf**) and/or **argumentsize** (**argsize**) options, the **memorysize** option sets these values for you. If you specify the **preprocessorbuffer** and/or **argumentsize** options, the values you specify override the values set by **memorysize**.

The following table lists the default values for `argumentsize` and `preprocessorbuffer` by `memorysize`. It also lists the buffer size and location of the compiler intermediate file.

<code>memsize</code>	<code>argsize</code>	<code>ppbuf</code>	Intermediate File Buffer	Debug Side File Buffer
<code>tiny</code>	127	1024	1024	2K
<code>small</code>	255	2048	4096	8K
<code>medium</code>	511	4096	8192	32K
<code>large</code>	1023	8192	no limit	64K
<code>huge</code>	4800	16384	no limit	128K

See also the descriptions of the `preprocessorbuffer` and `argumentsize` options.

Modified

tells the `sc` command to process only files that are out of date with respect to their output files. The default value is `nomodified`. The minimum acceptable abbreviation is `mod`.

This option is useful if you are compiling several files with a single `sc` command and only some of the files need to be recompiled. This option also works when you are generating prototypes or preprocessor output. If you also specify the `link` option, all object files are included in the link, even if their source files were not recompiled.

MultipleCharacterConstants

allows up to four bytes to appear within single quotes as a character constant. This option is included only for compatibility with previous releases of the compiler, and its use is not recommended. The minimum acceptable abbreviation is `mccons`. The default value is `nomccons`.

If you specify `mcccons`, a single constant of type `int` is generated. If fewer than four bytes are provided, they are padded on the left with zeroes, as in the following example:

```
#include <stdio.h>

void main(void)
{
    long l = 'abcd';
    long m = '\x01\x02\x03';
    printf("l=0x%08lx, m=0x%08lx\n", l, m);
}
```

This example program prints the following:

```
l=0x61626364, m=0x00010203
```

MultipleIncludes

tells the compiler to include header files that are included more than once. The default value is `multipleincludes`. The minimum acceptable abbreviation is `minc`.

This behavior is required by the ANSI Standard. However, many projects do not need this behavior, and specifying `nominc` can save compilation time.

Object=filename(s)

lists the object files that are to be linked into the program. The minimum acceptable abbreviation is `obj`. This option does not have a negative form.

The object files must have been created during a previous compilation. Do not specify object files that will be created by the same `sc` command in which you specify the `object` option. You can use AmigaDOS wildcard characters to specify filenames. To specify several filenames or wildcard patterns, separate each filename with a plus (+) sign or a comma (,). You can specify the `object` option as many times as necessary.

This option is ignored if you do not specify the `link` option. See also the descriptions of the `csource`, `library`, and `assembler` options.

ObjectLibrary=link-library-name

specifies that any resulting object files are to be placed in the named link library. The minimum acceptable abbreviation is **objlib**.

Do not specify this option if you also specify the **link** option.

ObjectName=file-or-directory-name

specifies the name of the file or directory to hold compiler and assembler output. The minimum acceptable abbreviation is **objname**.

If you are compiling or assembling only one C source file or assembly-language file, you can use this option to specify the file where you want the compiler or assembler output. If you are compiling or assembling more than one file, you can use this option to specify a directory where you want the output, and **sc** uses the same filename as the source file but replaces the extension with **.o**. The directory name must end with a forward slash (/) or a colon (:). If you do not use the **objname** option, **sc** uses the same filename with the extension of **.o**, but it places the files in the same directory as the source file.

OldPreprocessor

is provided for compatibility with pre-ANSI style preprocessors. The default value is **nooldpreprocessor**. The minimum acceptable abbreviation is **oldpp**.

The **oldpp** option does the following:

- ☐ allows old-style token pasting using comments
- ☐ substitutes values for parameters specified as quoted strings in macro definitions.

For example, suppose you have the following program:

```
#define FOO(bar) "bar"

void main(void)
{
    printf("%s\n", FOO(test));
}
```

If you compile this program with `oldpp`, the program prints `test`. If you compile this program with `nooldpp`, the program prints `bar`. To make this program work with the ANSI features and the `nooldpp` option, substitute the following for the `#define`:

```
#define F00(bar) #bar
```

As an additional example, suppose you have the following program:

```
#define F00(bar) foo_**/bar

void main(void)
{
    int foo_test;
    int foo_xxx;

    F00(test) = 10;
    F00(xxx) = 20;
    printf("%d %d\n", foo_test, foo_xxx);
}
```

If you compile this program with `oldpp`, the `#define` concatenates the text of the argument after the string `foo_` and the program prints `10 20`. If you compile this program with `nooldpp`, the program produces a compilation error. To make this program work with the ANSI features and the `nooldpp` option, substitute the following for the `#define`:

```
#define F00(bar) foo_##bar
```

OnError=x

tells `sc` what action to take if a C or assembly language source file generates an error. The minimum acceptable abbreviation is `onerr`. This option does not have a negative form.

You can specify one of the following:

stop or **s**

tells **sc** not to process any more source files. The default value is **stop**.

continue or **c**

tells **sc** to process the next source file.

If you specify the **link** option, but your program generates errors, your program is not linked even if you specify **onerr=continue**.

Optimize

enables the global optimizer (unless you specify **nooptglobal**) and peephole optimizer (unless you specify **nooptpeep**). The default value is **noptimize**. The minimum acceptable abbreviation is **opt**.

OptimizerAlias

disables type-based aliasing assumptions in the optimizer. The default value is **nooptimizeralias**. The minimum acceptable abbreviation is **optalias**.

If you specify **optimizeralias**, the global optimizer uses worst-case aliasing. Specifying **optimizeralias** can significantly reduce the amount of optimization that can be performed. This option is ignored if you do not specify the **optimize** and **optglobal** options.

OptimizerComplexity=n

defines the maximum complexity level of functions to be automatically inlined. The default value is 3. The minimum acceptable abbreviation is **optcomp**.

The parameter *n* represents the relative complexity of the function to be inlined and is a count of the number of discrete operations in the function. Try different values for this number until you get the results you want. Specify a value of 3 or higher. The higher the number, the more functions you can inline, but the size of your code will grow significantly as well.

If you specify **noptcomp**, no complexity-based inlining occurs. This option is ignored if you do not specify the **optimize**, **optglobal**, and **optimizerinline** options.

OptimizerDepth=*n*

defines the maximum nesting depth of automatically inlined functions. The default value is 3. The minimum acceptable abbreviation is **optdep**.

Specify a value of 3 or higher. This option is ignored if you do not specify the **optimize**, **optglobal**, and **optimizerinline** options.

OptimizerGlobal

enables the global optimizer. The default value is **optimizerglobal**. The minimum abbreviation is **optgo**. This option is ignored if you do not specify the **optimize** option.

OptimizerInline

allows inlining of functions, including functions defined with the **__inline** keyword. The default value is **optimizerinline**. The minimum acceptable abbreviation is **optinl**.

This option is ignored if you do not specify the **optglobal** and **optimize** options.

If you do not specify this option, the **optinlocal**, **optdepth**, **optcomplexity**, and **optrdepth** options are ignored.

OptimizerInLocal

inlines single-use static functions. The default value is **nooptimizerinlocal**. The minimum acceptable abbreviation is **optinlocal**.

This option is ignored if you do not specify the **optimize**, **optglobal**, and **optimizerinline** options.

OptimizerLoop

enables loop optimizations. The default value is **optimizerloop**. The minimum acceptable abbreviation is **optloop**.

This option is ignored if you do not specify the **optimize** and **optglobal** options.

OptimizerPeephole

enables the peephole optimizer. The default value is **optimizerpeephole**. The minimum acceptable abbreviation is **optpeep**.

This option is ignored if you do not specify the **optimize** option.

OptimizerRecurDepth=n

defines the maximum depth of recursion of automatically inlined functions. The default value is 3. The minimum acceptable abbreviation is **optrdep**.

Specify a value of 3 or higher. This option is ignored if you do not specify the **optimize**, **optglobal**, and **optimizerinline** options.

Optimizersize

disables optimizations that may sacrifice code size to save time. The default value is **nooptsize**. The minimum acceptable abbreviation is **optsize**.

This option is ignored if you do not specify the **optimize** and **optglobal** options. Do not specify both the **optimizersize** and **optimizertime** options.

Optimizertime

disables optimizations that may sacrifice time to save space. The default value is **nooptimizertime**. The minimum acceptable abbreviation is **opttime**.

This option is ignored if you do not specify the **optimize** and **optglobal** options. Do not specify both the **optimizersize** and **optimizertime** options.

Parameters=method

indicates how parameters should be passed. The minimum acceptable abbreviation is **parm**. This option does not have a negative form.

You can specify one of the following:

stack or s

indicates parameters should be passed on the stack. The default value is **stack**.

register or r

indicates parameters should be passed in registers.

both or b

generates a combination prolog that allows parameters to be passed on the stack or in registers.

If your program has prototypes for all routines, you should probably use **parameters=register** for increased efficiency. If you are placing code into a link library, specify **parms=both** so that your library functions accept registerized parameters and parameters that are passed on the stack.

If you write a function that has the same name as a library function, you need to add the `__regargs` keyword to your function or compile with the `parms=both` or `parms=register` option. For example, you may want to replace the SAS/C library function `malloc` with your own version of `malloc`. If you compile with the `parms=stack` option or define your version of `malloc` with the `__stdargs` keyword, then two versions of `malloc` are linked into your executable. If you use other SAS/C library functions that call `malloc`, these functions use the version of `malloc` in the SAS/C libraries. However, your functions that call `malloc` use your version of `malloc`. To make sure that all calls to `malloc` are using your version of `malloc`, define your version with `__regargs` or compile with the `parms=both` or `parms=register` option.

For information about passing parameters between C language functions and assembly-language functions, refer to Chapter 11, “Using Assembly Language with C Language,” in *SAS/C Development System User's Guide, Volume 2*.

Precision=precision

specifies the size of floating-point variables. You can specify `double` or `mixed`. The default value is `mixed`. The minimum acceptable abbreviation is `prec`. This option does not have a negative form.

If you specify `double`, data declared as `float` are treated as if they were declared as `double`. If you specify `mixed`, data declared as `float` and data declared as `double` are different sizes.

PreprocessorBuffer=n

sets the maximum number of bytes to which a macro can be expanded by the preprocessor. The default value is 8192 unless set differently by the `memorysize` option. The minimum acceptable abbreviation is `ppbuf`. This option does not have a negative form.

If a macro expands to greater than `n` bytes, the compiler issues an error message and aborts the compilation. See the description of the `memorysize` option for more information.

PreprocessorOnly

tells the compiler to run only the preprocessor on the C source files. The default value is `nopreprocessoronly`. The minimum acceptable abbreviation is `pponly`.

If you specify a filename with the **objectname** option, the compiler writes the output to the file you specify. If you do not specify an output filename, the compiler writes the output to the same filename as the source file but with the extension **.p**.

This option defines the preprocessor symbol **_PONLY**.

ProgramName=*output-module-name*

specifies the name of the executable module. The default value is the root name of the first source file specified in the **sc** command. The minimum acceptable abbreviation is **pname**. This option does not have a negative form.

This option is ignored if you do not specify the **link** option.

ResetOptions

resets all options to their default values. The minimum acceptable abbreviation is **resopt**. This option does not have a negative form.

If you specify this option as the first option in the **sc** command, it resets any option specified in the **scoptions** file. If you specify this option after other options or filenames in the **sc** command, it resets the options preceding it, including the filenames.

Saveds

generates code as if you had defined all functions in the source files with the **--saveds** keyword. The default value is **nosaveds**.

The **--saveds** keyword restores the near data pointer in register A4 at each function entry point. You should specify this option if you are compiling code that is used as an interrupt routine, called with the **AddTask** function, used in a shared library, or called from another process. This option does not work if your code is linked with the **cres.o** or **catchres.o** startup modules.

ShortIntegers

enables 16-bit integers. The default value is **noshortintegers**. The minimum acceptable abbreviation is **sint**.

If you specify **shortintegers**, types **int** and **short** are the same size. If you specify **noshortintegers**, types **int** and **long** are the same size. If you are running the assembler, this option defines the symbol **shortint**. This option also defines the preprocessor symbol **_SHORTINT**.

SmallCode

tells the linker to merge all **code** hunks into a single hunk. The default value is **nosmallcode**. The minimum acceptable abbreviation is **scode**.

This option is ignored if you do not specify the **link** option. See also the description of the **smallcode** linker option.

SmallData

tells the linker to merge all **data** and **bss** sections into a single hunk. The default value is **nosmalldata**. The minimum acceptable abbreviation is **sdata**.

This option is ignored if you do not specify the **link** option. See also the description of the **smalldata** linker option.

SourceIs=filename

sets the name of the C source file in the object file and in debugging information to the specified value instead of the actual name of the C source file. The minimum acceptable abbreviation is **srcis**.

You can use this option if you plan to rename or move the source file before using the debugger to debug your program. If you are compiling or assembling exactly one source file, **sourceis** can specify the name of the file to be placed into the debug information. If you are compiling or assembling multiple files, **sourceis** should specify a directory name (ending with a forward slash or colon).

StackCheck

generates stack overflow checking code at each function entry. The default value is **stackcheck**. The minimum acceptable abbreviation is **stkchk**.

When a program is about to run out of stack space, the program displays a requestor and terminates gracefully. For a complete description of the **stackcheck** option, see Chapter 11, "Using Amiga Specific Features of the SAS/C Language."

StackExtend

generates stack extension code at each function entry. The default value is **nostackext**. The minimum acceptable abbreviation is **stkext**.

When a program runs out of space in the current stack, a new stack is allocated, and your program continues to run. For a complete description of the **stackext** option, see Chapter 11, "Using Amiga Specific Features of the SAS/C Language."

StandardIO

specifies the use of `__main` instead of `__tinymain`. The default value is `stdio`. The minimum acceptable abbreviation is `stdio`.

To use `__tinymain`, specify `nostdio`. Using `__tinymain` makes your program require less space, but you cannot read from or send output to `stdin`, `stdout`, and `stderr`. This option is ignored if you do not specify the `link` option.

StartUp=module-name

specifies which startup module to use. The default value is `c` (for `c.o`). The minimum acceptable abbreviation is `strt`.

If the module name that you specify does not contain a colon (:), forward slash (/), or period (.), the compiler adds `lib:` to the beginning and `.o` to the end of the name you specify. If you do not want to compile with a startup module, specify `nostartup`. This option is ignored if you do not specify the `link` option.

Strict

enables a large number of diagnostics dealing with portability and questionable situations in your code. The default value is `nostrict`.

Specifying `strict` may produce warning messages for situations that are not a problem.

For more information on improving the portability of your code, see Chapter 13, “Writing Portable Code.”

StringsConst

tells the compiler to consider all string constants to be of type `const char *`. The default value is `nostringsconst`. The minimum acceptable abbreviation is `strcons`.

If you do not specify `stringsconst`, the compiler considers string constants to be of type `char *`. If you specify `stringsconst`, passing a string constant to a function that expects the type `char *` generates a warning message indicating that the function may modify the string constant.

StringMerge

merges all identical string constants in the C source file and places all strings, all data declared `static const`, and all auto initializers in the `code` section instead of the `data` section. The default value is `nostringmerge`. The minimum acceptable abbreviation is `strmer`.

If you specify **stringmerge**, the compiler examines each string constant defined in the code and checks for duplicates. If the compiler finds a duplicate string constant, it forces both references to refer to the same memory location.

If you specify **stringmerge** and you modify a string constant, the constant is modified in all locations. For example, suppose you have the following program:

```
#include <stdio.h>

void modifyme(char *);

void main(void)
{
    modifyme("Hello, World!\n");
    printf("Hello, World!\n");
}

void modifyme(char *msg)
{
    strcpy(msg, "Foobar\n");
}
```

If you compile the program with **stringmerge**, the program prints **Foobar**. If you compile the program with **nostringmerge**, the program prints **Hello, World!**.

The **stringmerge** option has two useful side effects. Because some data are placed in the code section:

- Less data needs to be placed into the near data section. If you have more than 64k of data, you can use the **stringmerge** option to try to reduce the amount of data and allow your code to continue to use the near data model.
- The string constants are addressed relative to the PC (program counter) instead of the beginning of the near data section. Therefore, it is possible to generate programs with no near or far data.

For more information on the code and near data sections, see Chapter 12, “How Does the Compiler Work?”

StripDebug

strips all debugging information from the final executable. The default value is **nostripdebug**. The minimum acceptable abbreviation is **stripdbg**.

This option is ignored if you do not specify the **link** option.

StructureEquivalence

tells the compiler not to issue messages if a pointer to one structure type is passed to a function when the function expects a pointer to a different type, if the type passed is equivalent to the type expected. The default value is **nostructureequivalence**. The minimum acceptable abbreviation is **streq**.

For information on using equivalent structures, see Chapter 11, “Using Amiga Specific Features of the SAS/C Language.”

To=filename

is included only for compatibility with the **slink** command. **to** is a synonym for the **programname** option. This option does not have a negative form.

Trigraph

specifies whether to use ANSI trigraphs. The default value is **notrigraph**. The minimum acceptable abbreviation is **trig**.

Specifying **trigraph** slows down the compiler.

Underscore

adds underscores to the beginning of all external names defined in any assembler source files assembled. The default value is **nounderscore**. The minimum acceptable abbreviation is **uscore**.

The underscores allow you to refer to these names in assembly language in the same way you do in C source files. This option is ignored if you do not specify any assembly-language files in the **sc** command. For more information, refer to Chapter 11, “Using Assembly Language with C Language,” in *SAS/C Development System User's Guide Volume 2*.

UnsignedChar

makes the default type of **char** variables **unsigned** instead of **signed**. The default value is **nounsignedchar**. The minimum acceptable abbreviation is **uchar**. This option defines the preprocessor symbol **_UNCHAR**.

UtilityLibrary

generates inline calls to the AmigaDOS 2.0 ROM-resident library **utility.library** to do integer multiplication and division instead of calling SAS/C library functions to do these operations. The default value is **nutilitylibrary**. The minimum acceptable abbreviation is **utllib**.

Specifying **utilitylibrary** makes your executable smaller and faster by taking advantage of 68020 instructions if they are available, but your program runs only under AmigaDOS 2.0. If you link using the **sc** command and the **utilitylibrary** and **link** options, **sc** defines all the SAS/C library integer conversion stub routines to stubs that call **utility.library**. This action prevents the SAS/C library routines that use integer conversion routines from using the SAS/C versions of the routines. If you link using the **slink** command, you need to specify the following **define** linker options:

```
define __CXM33=__UCXM33
define __CXD33=__UCXD33
define __CXM22=__UCXM22
define __CXD22=__UCXD22
```

These **define** options are in the file **LIB:utllib.with**, so you can link with the following command:

```
slink with LIB:utllib.with options
```

Verbose

displays messages about each stage of compiling and linking. The default value is **noverbose**.

Version

prints a banner containing the compiler version number and a copyright message. The default value is **version**. The minimum acceptable abbreviation is **ver**.

Warn=n

enables the specified compiler warning message. The minimum acceptable abbreviation is **wrn**. This option does not have a negative form.

You can specify **all** or **a** to enable all warning messages, or you can specify one or more message numbers to enable only those

messages. To specify several message numbers, separate each number with a plus (+) sign or a comma (,). You can specify the **warn** option as many times as necessary.

See also the description of the **error** and **ignore** options.

WarnVoidReturn

issues a warning message if a function declared as returning an integer actually returns no value. The default value is **warnvoidreturn**. The minimum acceptable abbreviation is **wvret**.

The **nowarnvoidreturn** option suppresses the **return value mismatch** warning message for functions declared as returning an integer, but that do not contain a **return** statement or that do not include an expression in the **return** statement.

With=filename

specifies the name of a file containing additional options. The additional options are read immediately, as if they were specified in the **sc** command at the position occupied by the **with** option. Options specified after the **with** option may override options specified in the **with** file. This option does not have a negative form. You can specify the **with** option as many times as necessary.

Note: Do not confuse this option with the **with** linker option.

XREF

is a synonym for the **xreference** option.

XReference

produces a cross-reference. The default value is **noxreference**. The minimum acceptable abbreviation is **xref**.

The compiler writes the output to the filename you specify with the **listfile** option. If you do not specify an output filename, the compiler uses the name of the first source file you specify but with the extension **.lst**.

XReferenceHeaders

generates a cross-reference of user header files. The default value is **xreferenceheaders**. The minimum acceptable abbreviation is **xhead**.

This option is ignored if you do not specify the **xreference** option.

XReferenceSystem

includes symbols defined in system header files in the cross-reference. The default value is **xreferencesystem**. The minimum acceptable abbreviation is **xsys**.

This option is ignored if you do not specify the **xreference** option.

Using Preprocessor Symbols Defined by the Compiler

The compiler automatically defines certain preprocessor symbols each time you run the compiler, and it defines other symbols only when you use specific compiler options.

Table 8.2 lists each of the symbols that the compiler automatically defines each time you run the compiler.

Table 8.2
*Preprocessor Symbols
Defined by the
Compiler*

Symbol	Value
_AMIGA	1
_M68000	1
__SASC	1
__SASC_60	1
__VERSION__	6
__REVISION__	0
__STDC__	1
__FILE__	<i>current filename</i>
__LINE__	<i>current line number</i>
__FUNC__	<i>current function</i>
__DATE__	<i>current date</i>
__TIME__	<i>current time</i>

Table 8.3 lists each of these preprocessor symbols defined when you use specific compiler options and the options that **#define** these symbols. These symbols are defined to 1 when you specify the corresponding option. For example, if you specify the **noansi** option, the **AMIGA** symbol is defined as follows:

```
#define AMIGA 1
```

Table 8.3
*Preprocessor Symbols
 Defined by Compiler
 Options*

Symbol	Compiler Options
AMIGA	noansi
LPTR	noansi and noshortint
SPTR	noansi and shortint
LATTICE	noansi
LATTICE_50	noansi
_M68010	cpu=68010 68020 68030 68040
_M68020	cpu=68020 68030 68040
_M68030	cpu=68030 68040
_M68040	cpu=68040
_M68881	math=68881
_FFP	math=FFP
_IEEE	math=IEEE
_SHORTINT	shortint
_UNSCHAR	unschar
_PPONLY	pponly
_GENPROTO	genproto
_MGST	makegst
_GST	gst

Linking Your Program

As described in Chapter 2, “Using Your SAS/C Development System,” you can link your program from the Shell by specifying the **link** option in the **sc** command or from the Workbench screen by specifying the **link** option in your **scoptions** file and double-clicking on the **Build** icon.

You can also compile and link your program in separate steps. If you do not specify the **link** option on the **sc** command line, you need to call the linker, **slink**, to link your program. Use the **slink** command only if you are limited by the **sc** command. For the majority of programs, linking with the **sc** command is much simpler and easier than linking with the **slink** command. If you link using the **sc** command, you can specify linker options with the **linkeropt** compiler option.

Entering The **slink** Command

After you have compiled your program, you can enter the **slink** command as follows:

```
slink object-files [options]
```

You must specify at least one object file. You probably also need to specify a startup module and one or more libraries.

Specifying Linker Options

The **slink** command accepts the following options:

addsym

generates **hunk__symbol** records for all externally-visible symbols in each object file and library module whether or not the object file was compiled with the **debug** compiler option. This option gives CodeProbe the location, but not the size or type, of any externally visible symbols in your program.

batch

sets the value of all undefined data symbols to 0 and all undefined code symbols to **__stub**. Normally, **slink** asks you to enter a value for each undefined symbol. However, if you specify **batch**, **slink** does not prompt you to enter values for undefined symbols.

If an undefined function is called, the library function **__stub** (**__stub** to the linker) is called instead. The library's version of **__stub** displays a requester telling you that an undefined routine was called and allows you to choose whether to abort or continue.

bufsize *n*

sets the I/O buffer size for **slink**. By default, **slink** buffers all object modules. Buffering requires more memory but reduces the time required to link your program. If you run out of memory while linking, try linking with **bufsize 4096**. This specification tells **slink** to buffer only one file at a time and to use a buffer size of 4096 bytes.

chip

specifies that all hunks are to be placed in chip memory regardless of the input object hunk specifications. However, if you specify **fast** or **chip** for the **datamem**, **codemem**, or **bssmem** compiler options, that value overrides this option.

The **chip** option is provided for compatibility with previous versions of the linker. It is recommended that you use the **datamem**, **codemem**, and **bssmem** compiler options instead.

define *symbol*[=*value*]

defines a symbol to be used in the linking process. You can use this option with the **prelink** option to force specific routines from the

libraries to be linked into your program even though your program does not contain any references to the routines.

Remember that linker symbols have an extra underscore added to the beginning of the symbol name. To refer to a function `foo` using the `define` option, you must specify `_foo`. To refer to registerized functions, add an at (@) sign to the beginning of the symbol name, as in `@foo`.

Note: Do not confuse this option with the `define` compiler option.

fancy

enters control characters to highlight and bold headings in the map file. The option is on by default. To disable the use of these control characters, specify the `plain` option. `fancy` is ignored if you do not specify the `map` option.

fast

specifies that all hunks are to be placed in fast or expansion memory regardless of the input object hunk specifications. However, if you specify `fast` or `chip` for the `datamem`, `codemem`, or `bssmem` compiler options, that value overrides this option.

The `fast` option is provided for compatibility with previous versions of the linker. It is recommended that you use the `datamem`, `codemem`, and `bssmem` compiler options instead.

faster

is included only for compatibility with the Commodore linker, `alink`.

from *object-filename(s)*

identifies the object files that are the primary input to the linker. You can specify the `from` option as many times as necessary in the `slink` command. If you specify the object files as the first items in the `slink` command, you do not need to use the `from` option. The `root` option is a synonym for this option.

The object files are copied to the root of the object module. You must specify at least one object file in the `slink` command.

fwidth *n*

specifies the filename width, in columns, in the map file. The default value is 16. This option is ignored if you do not specify the `map` option.

height *n*

specifies the total number of lines on a page in a map file. If you specify 0, the linker does not add form feed characters to the file. The default value is 55. This option is ignored if you do not specify the `map` option.

hwidth *n*

specifies the hunk name field width, in columns, in the map file. The default value is 8. This option is ignored if you do not specify the **map** option.

indent *n*

specifies the number of columns to indent on a line in the map file. The map file is indented *n* columns from the value specified by the **width** option. The default value is 0. This option is ignored if you do not specify the **map** option.

libfd *filename*

specifies the name of a function description (**.fd**) file. Use this option only if you are building a shared library.

For information on creating shared libraries, refer to *SAS/C Development System Library Reference*.

libprefix *prefix*

specifies the prefix you want added to the functions listed in the function description (**.fd**) file. The default prefix is an underscore (**_**). Use this option only if you are building a shared library. The **libprefix** option allows the library to call entry points within itself, without using the library base.

Make sure that the function declarations in your source code match the full name, including any specified prefix. For example, suppose your library has a function called **myfunc**. Your **.fd** file contains the following line:

```
myfunc(a)(d0)
```

If you specify a prefix of **_LIB**, you should declare the function in your source code as **LIBmyfunc**, as shown in the following example:

```
#pragma mylibbase myfunc 1e 001

__asm LIBmyfunc(register __d int a)
{
    return a+1;
}

void test(void)
{
    LIBmyfunc(1);    /* Call myfunc directly.          */
    myfunc(2);       /* Call myfunc through the library base. */
}
```

For information on creating shared libraries, refer to *SAS/C Development System Library Reference*.

library *link-library-filename(s)*

specifies the library files that you want the linker to search for modules referenced in your code. Only referenced modules from library files are included in the final executable module. You can abbreviate this option as **lib**.

librevision *n*

specifies a minor revision number of the library you are creating. If you do not specify a revision number, it defaults to 0. Use this option only if you are building a shared library.

For information on creating shared libraries, refer to *SAS/C Development System Library Reference*.

libversion *n*

sets the version number of the library you are creating. You can set the version number to any number from 0 to 255. The default is 1. Use this option only if you are building a shared library.

For information on creating shared libraries, refer to *SAS/C Development System Library Reference*.

map [*mapfile* [*map-option*[,]*map-option*...]]

tells **slink** to create a map file. The map file contains information on the size and location of all hunks and the value of all symbols.

If you do not specify a *mapfile*, the linker writes the map information to the file **executable.map**. If you specify a *mapfile*, you can also specify the following map file options:

- f** lists the hunks in each file
- h** lists hunks by size and originating function
- l** lists hunks by library function
- o** lists hunks in each overlay
- s** lists where symbols are defined
- x** creates a symbol cross-reference that lists where the symbols are defined and their definition.

maxhunk *n*

limits to *n* bytes the size of the hunks that **slink** creates when merging hunks. By default, there is no limit on hunk size.

noalvs

issues warning messages if **slink** generates an automatic link vector (**ALV**) instruction to resolve 16-bit program counter-relative references. Using this option stops **slink** from creating a non-relocatable module from what was intended to be relocatable code.

nodebug

is included for compatibility with previous release of the linker. Use the **stripdebug** option instead. You can abbreviate this option as **nd**.

onedata

tells the linker to merge all **data**, **bss**, and **chip** sections. Merging hunks may decrease the time required to load your program but may produce larger hunks that are difficult to scatter load.

overlay

identifies the start of an overlay tree. You should enter the **overlay** option and the list of files in each overlay in a **with** file, as follows:

```
overlay
overlay-1-filename(s)
*overlay-2-filename(s)
.
.
.
#
```

You must terminate the overlay tree with a line consisting of a single pound (#) sign. For more information, see “Using Overlays,” later in this chapter. See also the description of the **with** option.

ovlymgr filename

tells **slink** to use the filename you specify as the overlay manager instead of the default filename, **ovs.o**, which is contained in the libraries. The file you specify should consist of a single code hunk named **NTRYHUNK** and define the global symbol **_ovlyMgr**.

prelink

tells **slink** to produce an object module with symbol references and definitions still intact. You can then link with this object module to produce an executable module. This option is useful if you are developing a large application and changing only a single source module. Do not specify this option if you are using overlays.

plain

turns off the **fancy** option. This option tells the linker not to enter control characters for highlighting and bold in the map file. This option is ignored if you do not specify the **map** option.

pwidth n

specifies the width of program unit name field in the map file. The default value is 8. This option is ignored if you do not specify the **map** option.

quiet

suppresses all linker messages except error messages.

root *object-filename(s)*

specifies the object files that are the primary input to the linker. These object files are always copied to the root of the object module. You must specify at least one object file for the root. If the primary input files appear as the first option to **slink**, then the **root** keyword is optional and may be omitted. The **from** option is a synonym for **root**. You can specify the **root** option more than once. The files you specify with **root** are added to the list of files to be linked.

swidth *n*

specifies the width of the symbol names field in the map file. Default value is 8. This option is ignored if you do not specify the **map** option.

smallcode

tells the linker to merge all **code** hunks. You can abbreviate this option as **sc**.

smalldata

tells the linker to merge all **data** and **bss** sections. Merging hunks may decrease the time required to load your program, but may produce larger hunks that are difficult to scatter load. You can abbreviate this option as **sd**. The **smalldata** option does not merge chip data with non-chip data. To merge chip data also, specify the **onedata** option instead.

stripdebug

suppresses any **H_DEBUG** and **H_SYMBOL** debug information in the final executable file.

to *filename*

specifies the name of the final executable module. If you do not specify a **to** filename, the linker uses the filename of the second object module listed with the **from** option but drops any file extension. The first module is assumed to be startup code.

verbose

prints the names of each file as the file is processed.

verify *filename*

specifies a file to which you want the linker to write all messages. If you do not specify the **verify** option, the linker writes all messages to **stdout**. You can abbreviate this option as **ver**.

width *n*

sets the maximum line length for the map and cross reference listing files. The default value is 80. This option is ignored if you do not specify the **map** option.

with filename

specifies a file containing **slink** options. The linker processes the contents of **with** files as if you had specified the option in the **slink** command. You can specify the **with** option as many times as necessary. Also, you can enter **with** options in a **with** file.

The following is an example **with** file.

```
from lib:c.o vt100.o init.o window.o xmodem.o remote.o
    kermit.o script.o
library lib:sc.lib lib:amiga.lib
verbose
smallcode
smalldata
to vt100
```

Note: Do not confuse this option with the **with** compiler option.

xref filename

specifies a file to which you want the linker to write cross reference information. If you do not specify **xref**, but you specify the **map** option, the linker writes the cross reference information in the map file.

Note: Do not confuse this option with the **xref** compiler option.

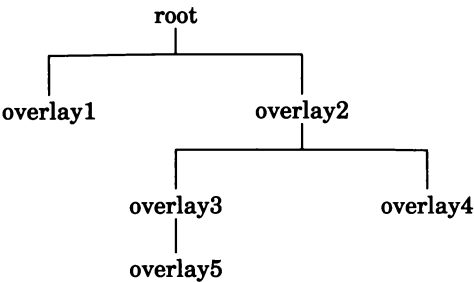
Using Overlays

Overlays are intended primarily for large applications (100k or more) that must run in a small amount of memory and leave more room for data. *Overlays* are groups of functions (and the data they require) that reside in memory only while in use. When the group in memory is no longer needed, the *overlay manager* reclaims the memory used by this group and loads in the next group of functions (and data).

Before deciding to use overlays, consider using multiple shared libraries instead. If memory is tight, the AmigaDOS system removes unused shared libraries, thereby giving you some of the benefits of overlays. On systems with more memory, the AmigaDOS system allows shared libraries to remain resident, which improves performance. Use overlays only in very tight memory situations.

The overlays are organized into an *overlay tree*. The overlay tree consists of a *root* node, which is always resident in memory, and one or more overlays. Figure 8.1 shows an example overlay tree.

Figure 8.1
An Example Overlay Tree



Overlays at the same level in the tree are mutually exclusive. In this example tree, the functions in overlays 1 and 2 are mutually exclusive, and the functions in overlays 3 and 4 are mutually exclusive. If any function in overlay 1 is running, overlay 2 cannot be loaded. However, if your program calls a function that is in overlay 2, the overlay manager reclaims the memory occupied by overlay 1 and loads overlay 2. Then, if your program calls a function in overlay 3, the overlay manager loads overlay 3. If overlay 3 is loaded, overlay 5 can be loaded if necessary, but overlay 4 cannot be loaded.

A function in an overlay can call functions in only certain other overlays. For example, a function in overlay 3 can call a function in overlays 5 and 2 and in the root node. The following list shows which functions the functions in each overlay of the example tree can call.

Functions in...	Can call functions in...
root	can call functions in overlays 1 and 2
1	can call functions in the root node
2	can call functions in the root node and overlays 3 and 4
3	can call functions in the root node and overlays 2 and 5
4	can call functions in the root node and overlay 2
5	can call functions in the root node and overlays 2 and 3.

As you can see, applications best suited to overlays are those that are modular and have a definite calling hierarchy among the functions. You must determine which functions should be in each overlay. Decide which functions can operate independently of others. The root node must

contain your main program and should contain any functions that are called frequently or by many other functions in the program. The root node usually contains the libraries also. In general, the more frequently you call a function, the closer it should be to the root node.

You can have up to 50 overlays. Your tree could have all overlays hanging from the root node, or your tree could be organized as one long chain of overlays. Organize the overlays of your tree efficiently; that is, according to the calling hierarchy of your program. If you create an inefficient tree, your program may not run efficiently because the overlays will have to be swapped more often. If you create an incorrect tree, **slink** issues a warning message.

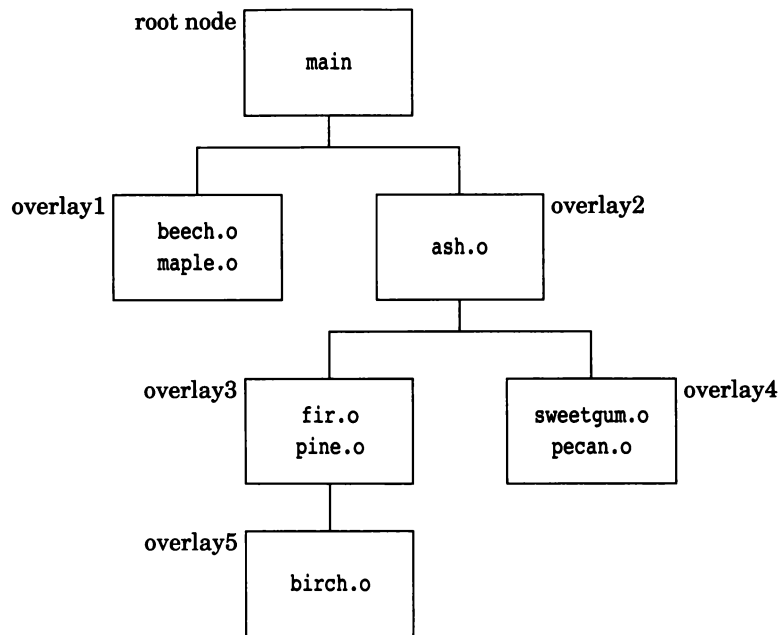
To use overlays, you should enter the **overlay** option and the overlay filenames in a **with** file, as follows:

```
overlay
overlay-1-filenames
*overlay-2-filenames
.
.
.
#
```

List the filenames for each overlay on a separate line, and separate the filenames with plus (+) signs. If you cannot fit all of the filenames on one line, end the line with a plus sign, and enter the remaining filenames on the next line. Indicate the depth in the tree of the overlay by entering the appropriate number of asterisks at the beginning of the line. The overlays immediately beneath the root node are at depth 1, but you should not list them with asterisks. For overlays at depth 2, begin the line(s) with one asterisk; for overlays at depth 3, begin the line(s) with two asterisks, and so on. Enter an overlay immediately after the overlay to which it is subordinate. Terminate the overlay tree with a line consisting of a single pound (#) sign.

For example, Figure 8.2 shows the same overlay tree as in Figure 8.1, but figure 8.2 shows which object files comprise each overlay.

Figure 8.2
*Example Overlay Tree
 with Filenames*



The `with` file for the example in Figure 8.2 looks like this:

```

overlay
beech.o+maple.o
ash.o
*fir.o+pine.o
**birch.o
*sweetgum.o+pecan.o
#

```

Suppose you have three files: `hw.c`, `hello.c`, and `world.c`. The file `hw.c` contains the following functions:

```

#include <stdio.h>

int main(void)
{
    Hello(void);
    World(void);
    return(0);
}

```

```
void SayWord(s)
    char *s;
{
    printf("%s", s);
}
```

The file `hello.c` contains only one function:

```
void Hello(void)
{
    SayWord("\nHello, ");
}
```

`world.c` also contains only one function:

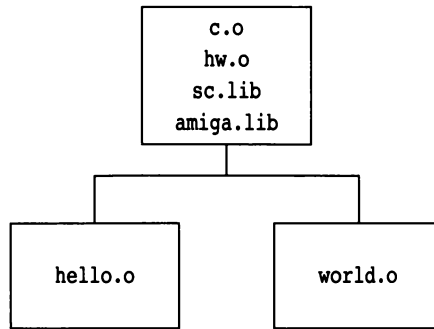
```
void World(void)
{
    SayWord("World!\n");
}
```

After you compiled these three files, you have the three object files `hw.o`, `hello.o`, and `world.o`. To link this application, you can create a `with` file that contains all of the link options you need, including overlay specifications.

```
from lib:c.o hw.o
overlay
hello.o
world.o
#
library lib:sc.lib lib:amiga.lib
map fhlosx
to hw
```

`slink` evaluates the `with` file, creates an executable module named `hw`, and creates a map file named `hw.map`. The executable module consists of a root node and two overlays. The root node contains all of the code and data for `c.o`, `hw.o`, `sc.lib`, and `amiga.lib`. One overlay contains the code and data for `hello.o`, and the other overlay contains the code and data for `world.o`. Figure 8.3 shows the overlay tree for this application.

Figure 8.3
Overlay Tree for hw



You can run this application in the same way you run an application that does not use overlays. The overlay manager loads overlays as necessary.

Specifying Compiler Options with Overlays

Using overlays modifies the behavior of some compiler options. The following list describes each of the options affected by overlays.

data=near

places all data not declared with `__far` or `__chip` in the `near` data section. If you are using overlays, the near data segments of all files compiled with `data=near` are moved to the root node. Therefore, you cannot statically initialize near data items into code hunks that are not in the root node. For example, you cannot initialize a character pointer to a constant string in an overlay node if you specify the `stringmerge` option.

smallcode

merges all `code` hunks into a single hunk. If you are using overlays and you specify `smallcode`, `slink` merges as much of the code as possible without crossing overlay boundaries. The linker does not move code out of the overlay in which it was defined.

smalldata

merges all `data` and `bss` hunks into a single hunk. The near data section is always placed in the root node. However, `slink` does not merge hunks that are in overlays that are not named `__MERGED` into the near data section.

stringmerge

places string constants into the code section, and the strings themselves are overlaid if the module they are used in is overlaid. Therefore, the value of the string constant may change as its overlay node is swapped out. For example, a function may return a string

constant or assign it to an external variable. It is recommended that you do not compile with **stringmerge** if you are using overlays.

For some applications, you may want to replace the overlay manager with a custom overlay manager. For example, you can write an overlay manager to support function calls across mutually exclusive overlays. This section describes the necessary attributes of an overlay manager.

The overlay manager must contain one code hunk named **NTRYHUNK** and define the global symbol **_ovlyMgr**.

```

        section    NTRYHUNK,code
        xdef       ovlyMgr

First:
        bra        FirstModule

ovlyMgr:

        ;Overlay manager code goes here.

FirstModule:
        lea        First(PC),a3    ; pointer to start of module
        move.l     -4(a3),d7        ; BPTR to next seglist entry
        asl.l      #4,d7            ; make into APTR
        move.l     d7,a3
        jmp        4(a3)            ; jump to code in next hunk
    
```

Assemble this code with the **-u** assembler option so that the assembler adds the necessary underscores to all global symbols.

The linker makes **NTRYHUNK** the first hunk in the executable and does not merge this hunk with any other hunks. Therefore, **NTRYHUNK** must also contain a valid entry point to the program and properly transfer control to the rest of the program. The first instruction in the preceding code branches to the symbol **FirstModule**, where the seglist is decoded, and a jump instruction transfers control to the next hunk in the segment list.

For more information, examine the file **ovs.a** in the **source** directory. This file contains the source to the default overlay manager.

9 Running Your Program from the Workbench™ Screen

135	<i>Introduction</i>
135	<i>Working with argc and argv</i>
135	<i>Running From the Shell</i>
136	<i>Running From the Workbench Screen</i>
137	<i>Managing the Standard I/O Window</i>
137	<i>Eliminating the Window</i>
137	<i>Changing the Window Attributes</i>

Introduction

If you run your program by double-clicking its icon from the Workbench screen, your program runs under a different environment than programs that are run from the Shell. Specifically:

- If not handled correctly, the command line arguments `argc` and `argv` receive different values that may cause unwary programs to crash.
- If your program uses `stdin`, `stdout`, or `stderr`, the library routine `__main` opens a console window to which your program writes output and from which it reads input.

The following sections describe how to manage these unusual situations.

Working with argc and argv

Programs that run from the Workbench screen receive a different environment than those run from the Shell. The following sections describe the different environments.

Running From the Shell

If you run your program from the Shell, your program receives two arguments, `argc` and `argv`. `argc` is an integer guaranteed to have a value of at least 1. Unless you modify `__main` as described in Chapter 10, “Using Startup Modules, Autoinitialization, and

Autotermination Functions,” `argv` is an array of 32 character pointers that point to the different items on the command line:

`argv[0]`

is a pointer to the name of the command that was executed.

`argv[1]` to `argv[n]`

are pointers to the arguments entered after the command. The number *n* represents the number of arguments specified. The maximum number of arguments is 30.

`argv[n+1]` to `argv[31]`

are `NULL`.

Running From the Workbench Screen

If you run your program from the Workbench screen, the startup code sets up `argv` and `argc` in a different way than if your program is run from the Shell. `argc` is set to 0 (zero) to give you a quick way to check in your program if you are running from the Workbench screen, and `argv` is a pointer to a `struct WBStartup` instead of an array of character pointers. The definition of `struct WBStartup` is in the header file `workbench/startup.h`.

To eliminate any problems caused by running your program from the Workbench environment, structure your program as follows:

```
int main(int argc, char *argv[])
{
    struct WBStartup *startup;
    if(argc == 0)
    {
        /* The program was started from the Workbench.      */
        /* Use the WBStartup message instead of argc and argv. */
        startup = (struct WBStartup *)argv;
        ...
    }
    else
    {
        /* The program was started from the Shell. */
        /* Use argc and argv as usual.             */
        ...
    }
}
```

For more information on `main`, refer to *SAS/C Development System Library Reference, Version 6.0*.

Managing the Standard I/O Window

When you run a program from the Workbench screen, the compiler opens a standard I/O window in which your program can read from and write to `stdin`, `stdout`, and `stderr`. You can delete the window, change its size, or add a title.

The window is opened by the library routine `__main`, which is automatically run before your main routine if you link with a startup module.

Eliminating the Window

If your application does not use `stdin`, `stdout`, or `stderr`, you can use an alternate routine that does not open the window:

- You can use the alternate routine `__tinymain`. If you link your program using the `link` compiler option, specify the `nostdio` option to delete the window. If you run the linker by specifying the `slink` command, specify the `define` linker option as follows:

```
define __main=__tinymain
```

Use this `define` option regardless of any other compiler options you are using. This option works even if you are using registerized parameters.

- You can write your own routine.
- You can copy `_main.c` from the `sc:source` directory and modify it for your application.

► **Caution** *Your program may crash.*

If you attempt to use any function that refers to `stdin`, `stdout`, or `stderr` in a program run from the Workbench screen after deleting the standard I/O window, your program will crash. Such functions include `printf` and `scanf` (but not `fprintf` or `fscanf`). ▲

Changing the Window Attributes

To keep the window but change its attributes you need to declare the external variable `__stdiowin` in your program. Normally, `__main` gets the definition of `__stdiowin` from the library with which you link your program. However, if you declare `__stdiowin` in your program, `__main` uses the definition in your program.

This variable is declared in the SAS/C Libraries as follows:

```
char __stdiowin[] = "CON:10/10/320/80/";
```

This specification opens a console window starting at location (10,10) that is 320 pixels wide and 80 pixels high. For more information on console specifications, refer to *The AmigaDOS Manual, 3rd Edition* (Commodore-Amiga, Inc. 1991).

To change the window attributes, include a line similar to the previous line in any C source file in your project. Declare this variable external to any function, and be sure to statically initialize the variable. (Any changes made to this external variable after your program starts do not affect the window.) For example, to make the initial window 600 pixels wide and 100 pixels tall and add a window title of "My Window," you would enter the following line:

```
char __stdiowin[] = "CON:10/10/600/100/My Window";
```

If you are running under AmigaDOS 2.0, `__main` appends the contents of the array `__stdiov37` to the `__stdiowin` variable before opening the window. This array contains keywords that control other aspects of the window and is defined in the SAS/C Libraries as follows:

```
char __stdiov37[] = "/AUTO/CLOSE/WAIT";
```

These keywords do the following to your startup window:

- `/AUTO` indicates that you do not want the window to open unless output is written to or input is read from the window.
- `/CLOSE` adds a `Close` gadget to the window.
- `/WAIT` specifies that you do not want the window to close until the user clicks on the `Close` gadget or presses Ctrl-\ (End-of-File).

For more information on `CON:` keywords, refer to *The AmigaDOS Manual, 3rd Edition* (Commodore-Amiga, Inc. 1991).

To change these keywords or add additional AmigaDOS 2.0 keywords, include a line similar to the previous line in any C source file in your project. Declare this variable external to any function, and be sure to statically initialize the variable. (Any changes made to this external variable after your program starts do not affect the window.) The keywords you specify are appended to `__stdiowin` only if your program is running under AmigaDOS 2.0. If your program is run under earlier versions of the AmigaDOS operating system, you get a normal console window.

Note: You must declare `__stdiowin` and `__stdiov37` as external variables exactly as shown. Do not declare them as follows:

```
char *__stdiowin = "specification"; /* WRONG! */
```

10 Using Startup Modules, Autoinitialization, and Autotermination Functions

139	<i>Introduction</i>
140	<i>Understanding Startup Modules</i>
140	<i>Modifying — _main</i>
141	<i>Linking with a Startup Module</i>
142	<i>Creating Standard Programs</i>
142	<i>Creating Detachable (Load and Stay Resident) Programs</i>
143	<i>Creating Residentable Programs</i>
144	<i>Creating Shared Libraries</i>
145	<i>Writing Applications without a Startup Module</i>
145	<i>Using Autoinitialization and Autotermination Functions</i>
147	<i>Initializing System Library Bases</i>

Introduction

This chapter describes how to use startup modules, autoinitialization functions, and autotermination functions. Specifically, it describes the following:

- the function of a startup module
- how to link your program with a startup module
- how to use the startup modules distributed with the SAS/C Development System
- how to write a program that does not use a startup module
- how to specify autoinitialization and autotermination functions.

Understanding Startup Modules

A startup module initializes the environment in which your program runs and performs certain cleanup tasks after your program has finished running. The SAS/C Development System provides six startup modules:

- `c.o` is for use with standard programs.
- `cres.o` is for use with residentable programs.
- `cback.o` is for use with detachable programs.
- `catch.o` is for use when you are debugging standard programs.
- `catchres.o` is for use when you are debugging residentable programs.

The specific things that the startup module does depends on the startup module with which you link your program and on whether you run your program from the Shell or the Workbench screen. However, all startup modules perform some of the same tasks. For example, each startup module does the following:

- ☐ initializes the near **BSS** section.
- ☐ gets startup information from the Workbench screen (if invoked from the Workbench screen).
- ☐ initializes **DOSBase** and **SysBase**.
- ☐ marks the bottom of the stack.
- ☐ calls `__fpinit`, which looks for autoinitialization functions.
- ☐ calls `__main`, which calls your program. If your program requires more than 30 command line arguments, you need to modify the default `__main` function as described under “Modifying `__main`.”
- ☐ calls `__fpterm`, which also looks for autotermination functions.

Modifying `__main`

The default `__main` function must convert the incoming command line to the `(argc, argv)` format required by the ANSI Standard. `__main` performs this conversion by using a local array with 32 elements to represent `argv`. One element is used for the program name, and the ANSI Standard requires at least one **NULL** element at the end. Therefore, you are limited to 30 arguments on the command line of your program. If you need more arguments on the command line, you have two choices:

- ☐ If you only want to raise the argument limit, copy `sc:source/_main.c` to your project directory. Change the `#define` statement for the symbol **MAXARG** to the new maximum number of arguments. Then, recompile `_main.c` and link it into your program.

- If you need an unlimited number of arguments, copy `sc:source/_main.c` to your project directory. Modify the source code to allocate an array of pointers to use for `argv`. Make sure you use the `malloc` or `calloc` function to allocate the memory. These functions will free the memory correctly when your program exits.

Linking with a Startup Module

To use a startup module, you must link your program with it. You can link your program by using the `link` option on the `sc` command or by entering the `slink` command.

You can specify the `link` and, if necessary, the `startup` option on the `sc` command line. If you specify only the `link` option, `sc` tells the linker to link your program with `c.o`. If you want to link with another startup module, specify the module name with the `startup` option. If the module name that you specify does not contain a colon (:), forward slash (/), or period (.), `sc` adds the `LIB:` prefix and `.o` extension to the filename that you specify. For example, to link with `cres.o`, you can enter the `sc` command as follows:

```
sc startup=cres link C-source-file
```

To specify a startup module with the `slink` command, enter the `slink` command in the following format:

```
slink startup-module+obj-module to executable
lib library-files
```

To use some startup modules, you need to declare certain variables in your program. The following sections describe how to use startup modules for the type of program you want to run.

Programs linked with any startup module except `cback.o` run under the process in which they were invoked. When you enter the name of an executable module at the Shell prompt, that program runs in that Shell's process and uses the Shell's process number and stack (unless you specify a value for `__stack` and the Shell's stack is smaller than the specified value). Therefore, you cannot enter commands into or close the Shell from which the program was invoked until the program finishes running.

Note: Even if you use the `run` command to run the program, you cannot close the window from which the program is run unless you redirect the input and output as in the following example:

```
run >nil: <nil: program-name
```


Creating Standard Programs

For most of your programs, you can use the default startup module, `c.o`.

If you are debugging a standard program, you can use the `catch.o` startup module to capture information if your program terminates abnormally. If you link with `catch.o` and your program terminates abnormally, `catch.o` asks you if you want to create a snapshot file. A snapshot file contains traceback information for your program such as register values, symbol cross references, and stack information.

► **Caution** *Creating this snapshot file may corrupt your hard drive.*
Do not ship commercial products linked with `catch.o`. ▲

You can use the `tb` utility to process this snapshot file and display the traceback information. For more information about processing snapshot files, refer to the description of the `tb` utility in *SAS/C Development System User's Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0*.

Creating Detachable (Load and Stay Resident) Programs

A *detachable program* is a program that runs in the background. It does not run as part of the originating Shell's process. You can continue to enter commands into, or even close, the Shell from which the program was started without affecting the program.

You can use the startup module `cback.o` to create detachable programs. When you link with `cback.o`, your program allocates a new stack and launches itself as a new process. For an example of a program that links with `cback.o`, see the directory `sc:examples/cback`.

You should use `cback.o` for any program that:

- does not send output to the Shell from which it was invoked (except an initial banner)
- needs to be memory resident.

To use `cback.o`, you add the following external definitions to your main program if your program requires values other than the defaults:

```
long __stack = amount;
```

allocates stack space for the created task. Many detachable programs only need a stack size of 4000 bytes (the default). However, you can change this figure to suit your application.

```
char *__procname = "program-name";
```

specifies the name of the process to be created. The default value is `Background_Process`.

```
long __priority = priority;
```

sets the priority of the process to be created. For most programs, you should set the task priority to 0, which is the default. However, you can change this figure to suit your application.

```
long __BackGroundIO = value;
```

indicates whether output will be sent to the Shell from which the program was invoked. The default value is 0, which tells `cback.o` that no output is sent to the originating Shell. Setting this value to 1 tells `cback.o` to set `_Backstdout` to point to the Shell.

```
extern BPTR _Backstdout;
```

contains the file handle of the originating Shell (if `__BackGroundIO` was set to 1).

`_Backstdout` is an AmigaDOS file handle and, therefore, you must use the AmigaDOS function `Write` instead of the SAS/C functions such as `write` or `fprintf` to write to this file.

You must close this file with the AmigaDOS `Close` function when you have finished writing to the Shell window. If you do not close `_Backstdout`, the originating Shell cannot close.

If your program creates a resident task that needs to print messages in the Shell, your program should pass `_Backstdout` to this task.

If the `_Backstdout` file handle is `NULL`, your program cannot send output to the window. The handle may be `NULL` if the program was invoked from the Workbench screen or from another program that did not have a Shell open.

If you want to make sure that only one copy of your program is running in the background, have your program create a global message port with a unique name. Subsequent invocations of your program should use the exec function `FindPort` to see if the port exists and, if the port exists, pass any commands to the running program through the message port and exit immediately. See the directory `sc:examples/cback` for sample code.

Programs linked with `cback.o` cannot be made resident with the AmigaDOS `resident` command.

Creating Residentable Programs

A *resident program* is a program that remains loaded in RAM. You do not have to reload a resident program each time you want to run it.

To create a resident program, follow these rules:

- ☐ Compile with the `data=near` option (the default).
- ☐ Make sure that all external data declared with the `__far` or `__chip` keywords are read-only.
- ☐ Use `#pragma` statements instead of the linker stubs in `amiga.lib` when possible because `amiga.lib` uses absolute addressing. If required, you can link with `amiga.lib` if the linker does not generate any warnings about references to absolute data.
- ☐ Link your program with `cres.o` instead of `c.o`.
- ☐ Use the AmigaDOS `resident` command to add the program to the resident list maintained by the Shell.

A resident program must be re-entrant and re-executable, and it cannot write to far data. For each invocation of a resident program, `cres.o` copies the near data section, so the program can read and write to near data. If you link your program with `cres.o` and your program refers to `far` or `chip` data, the linker issues warnings. If your program reads this data only, you can ignore the warnings.

You cannot use the `__saveds` keyword or compile with the `saveds` option if you link with `cres.o`.

Each time the program is run, a new near data segment is created, initialized data are copied into the new segment, and relocations are performed inside the segment. When the program finishes, the segment is released.

`slink` creates the following symbols when you link your program:

- `_LinkerDB` points to the near data section.
- `RESBASE` specifies the offset of `_LinkerDB` in the near data section (0 or `0x8000`).
- `RESLEN` specifies the total number of bytes of initialized and BSS data.
- `NEWDATAL` specifies the number of longwords of initialized data.

For more information about resident programs, refer to the description of the AmigaDOS `resident` command in *Using the System Software* (Commodore-Amiga, Inc. 1990) or see the example in `sc:examples/cres`.

If you are debugging a residentable program, you can use the `catchres.o` startup module to capture information if your program terminates abnormally. If you link with `catchres.o` and your program terminates abnormally, `catchres.o` asks you if you want to create a snapshot file. A snapshot file contains traceback information for your program such as register values, symbol cross references, and stack information.

► **Caution** *Creating this snapshot file may corrupt your hard drive.*
Do not ship commercial products linked with `catchres.o`. ▲

You can use the `tb` utility to process this snapshot file and display the traceback information. For more information about processing snapshot files, refer to the description of the `tb` utility in *SAS/C Development System User's Guide, Volume 2*.

Creating Shared Libraries

You can use the modules `libent.o`, `libinit.o`, and `libinitr.o` to create shared libraries. For information on creating shared libraries, refer to *SAS/C Development System Library Reference, Version 6.0*.

Writing Applications without a Startup Module

If you do not want to link your program with a startup module, follow these guidelines when writing and compiling your program:

- Your program must initialize `DOSBase` and `SysBase`, if your program makes any calls to `dos.library` and `exec.library`, respectively. For additional information, see “Initializing System Library Bases,” later in this chapter.
- Do not use any SAS/C I/O functions such as `fopen`, `read`, `printf`, and so on.
- Add the `__saveds` keyword to the first function in the first object module in your program.
- Compile your program with the `nostackcheck` option.
- Be aware that the uninitialized near data are not initialized to 0.
- Register A0 points to the command line entered by the user. You can parse the arguments from A0 or, under AmigaDOS 2.0, call `ReadArgs` to get the arguments.
- Your program will start executing at the first function in the first module specified on your `slink` or `sc` command line. This function should be declared with the `__saveds` keyword.
- Unless you write code to reply to the Workbench startup message, your program will not run from the Workbench screen without crashing the machine.

For an example, see `sc:examples/nostartup`.

Using Autoinitialization and Autotermination Functions

If you link with one of the startup modules provided by the SAS/C Development System, you can designate functions to be called before your program starts (*autoinitialization functions*) and after it terminates (*autotermination functions*).

To designate an autoinitialization function, begin the function name with `__STI`. To designate an autotermination function, begin the function name with `__STD`. The linker searches the functions defined in your program for functions whose names start with the `__STI` or `__STD`. All functions starting with `__STI` are assumed to be autoinitialization functions and are called from the startup code before `__main` is called. All functions starting with `__STD` are assumed to be autotermination functions and are called after your program exits.

Autoinitialization and autotermination functions take no parameters and return no values, as shown in the following example:

```
void _STImyinit(void)
{
    .
    /* your code here */
    .
}

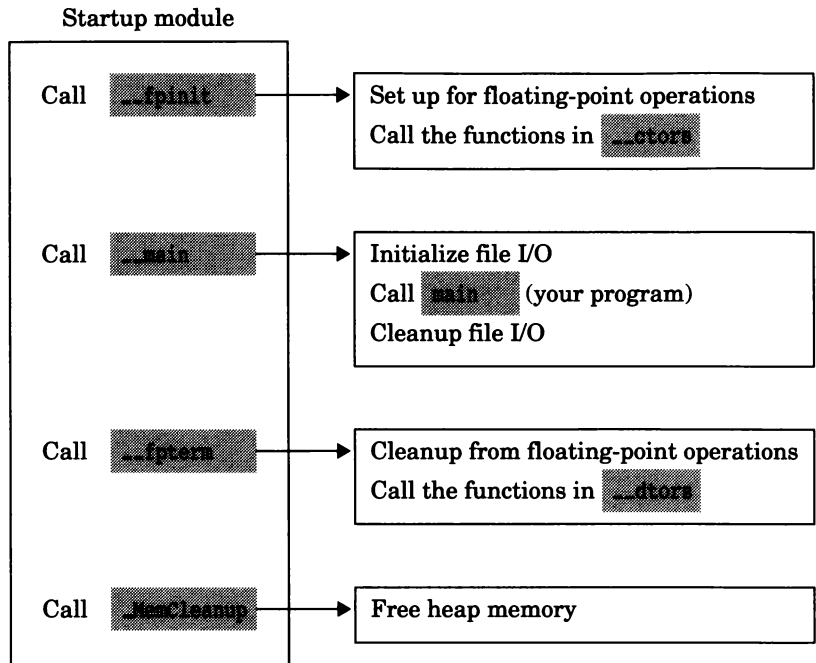
void _STDmyterm(void)
{
    .
    /* your code here */
    .
}
```

If the linker finds at least one autoinitialization function, it creates an array of function pointers containing one pointer for each autoinitialization function. The array is named `__ctors` (for “constructors”), and its last entry is always `NULL`. The `__fpinit` function looks through this array and calls each autoinitialization function before calling `__main`.

Similarly, if the linker finds at least one autotermination function, it creates an array of function pointers called `__dtors` (for “destructors”) containing one pointer for each autotermination function. The array’s last entry is always `NULL`. The `__fpterm` function looks through this array and calls each autotermination function after your program exits.

The following figure shows the order in which modules are called when you run your program.

Figure 10.1
Startup Modules,
Autoinitialization
Functions, and
Autotermination
Functions



If `slink` does not find any autoinitialization or autotermination functions, it substitutes a longword zero for any references to `__ctors` or `__dtors`. If you write your own startup module, check that `__ctors` and `__dtors` are not `NULL` before accessing the array.

Do not use level 1 or 2 I/O functions (such as `fopen` and `open`) in your autoinitialization functions because they have not yet been initialized. However, you can use AmigaDOS I/O functions (such as `Open`). Similarly, you cannot use level 1 and 2 I/O functions in your autotermination functions because all level 1 and 2 files have already been closed.

You can use the ANSI standard memory allocation routines (such as `malloc`, `free`, and `calloc`) in autoinitialization and autotermination functions. All memory allocated with these routines and not yet freed in your program is still valid in your autotermination functions.

Initializing System Library Bases

If you declare but do not define a system library base in your own code, the library base is automatically initialized. The autoinitialization code calls `OpenLibrary` to initialize the library base, and the

autotermination code calls `CloseLibrary` to close the library. In the following example, `IntuitionBase` is automatically initialized.

```
#include <proto/intuition.h>

/* The following line declares IntuitionBase but does */
/* not define it. This line is not required, because */
/* proto/intuition.h also declares IntuitionBase. In */
/* this program, IntuitionBase is automatically */
/* initialized. */

extern struct IntuitionBase *IntuitionBase;

int main(void)
{
    struct Window *w;

    w = OpenWindow(...);
    ...
}
```

The following table lists the system libraries whose bases are automatically initialized by the autoinitialization code.

Table 10.1
*Library Bases for
.library Files*

Name	Base
asl.library (2.0)	AslBase
commodities.library (2.0)	CxBase
diskfont.library	DiskfontBase
expansion.library	ExpansionBase
gadtools.library (2.0)	GadToolsBase
graphics.library	GfxBase
icon.library	IconBase
iffparse.library	IFFParseBase
intuition.library	IntuitionBase

(continued)

Table 10.1
(continued)

Name	Base
<code>keymap.library</code>	<code>KeymapBase</code>
<code>layers.library</code>	<code>LayersBase</code>
<code>mathffp.library</code>	<code>MathBase</code>
<code>mathieeedoubbas.library</code>	<code>MathIeeeDoubBasBase</code>
<code>mathieeedoubtrans.library</code>	<code>MathIeeeDoubTransBase</code>
<code>mathieeesingbas.library</code>	<code>MathIeeeSingBasBase</code>
<code>mathieeesingtrans.library</code>	<code>MathIeeeSingTransBase</code>
<code>mathtrans.library</code>	<code>MathTransBase</code>
<code>translator.library</code>	<code>TranslatorBase</code>
<code>utility.library</code>	<code>UtilityBase</code>
<code>workbench.library</code>	<code>WorkbenchBase</code>

Note: `DOSBase` and `SysBase` are initialized by the startup code even if you define them.

As stated before, a system library base is not initialized if you define the library base in your code (except `DOSBase` and `SysBase`). For example, the following code defines the library base and, therefore, the base is not automatically initialized.

```
#include <proto/intuition.h>

/* The following line defines IntuitionBase, so */
/* it is not automatically initialized.          */

struct IntuitionBase *IntuitionBase;
```



```

int main(void)
{
    struct Window *w;

    w = OpenWindow(...); /* Crash! IntuitionBase in NULL */
    ...
}

```

For more information on defining library bases, refer to *SAS/C Development System Library Reference*.

The autoinitialization functions pass the value of the external long integer `__OSlibversion` to `OpenLibrary`. If your program runs only under a specific version of the operating system, declare this variable in your code and initialize it as necessary. For example, if your program requires the AmigaDOS operating system, Version 2.0 (which is library revision 37), you can enter the following line in your program external to any function:

```
long __OSlibversion = 37;
```

The following table lists the library revision numbers for each version of the operating system.

OS Version	Library Revision
1.2	33
1.3	34
2.0	36 (Preliminary 2.0)
2.04	37

If the autoinitialized libraries cannot be opened, the library function `__autoopenfail` is called. The version of `__autoopenfail` in the libraries prints a message indicating which library could not be opened and then terminates your program. You can replace this function by declaring it in your code as follows:

```

void __autoopenfail(char *lib)
{

```

```
    /* your code here */  
}
```

The `lib` parameter is the name of the library that failed to open. If you want your program to terminate after the call to `__autoopenfail`, the last action of `__autoopenfail` should be to call `_XCEXIT`. If `__autoopenfail` returns without calling `_XCEXIT`, the startup proceeds normally.

The source code to the default `__autoopenfail` function is in `sc:source/autoopenfail.c`. As an additional example, `sc:source/intuitlib.c` contains the autoinitialization and autotermination code for `intuition.library`.

11 Using Amiga® Specific Features of the SAS/C® Language

153	<i>Introduction</i>
154	<i>Using Special Keywords</i>
155	<i>Using <code>__aligned</code></i>
156	<i>Using <code>__chip</code>, <code>__far</code>, and <code>__near</code></i>
157	<i>Using <code>__interrupt</code></i>
157	<i>Using <code>__asm</code>, <code>__regargs</code>, And <code>__stdargs</code></i>
158	<i>Using <code>__saveds</code></i>
159	<i>Using <code>__inline</code></i>
160	<i>Managing Stack Space</i>
161	<i>Using <code>#pragma</code> Statements</i>
163	<i>Using Unnamed Unions</i>
164	<i>Using Implicit Structure References</i>
164	<i>Using Equivalent Structures</i>
166	<i>Using Zero-Length Arrays</i>
167	<i>Specifying the Size of Enumerated Types</i>
168	<i>Using the <code>sizeof</code> and Comma Operators in Preprocessor Directives</i>
168	<i>Using C++ Style and Nested Comments</i>
168	<i>Using National Characters in Variable Names</i>
168	<i>Using Common Model External Data</i>

Introduction

This chapter describes features of the SAS/C Compiler that are not defined by the ANSI Standard. Specifically, the compiler allows you to do the following:

- ☐ use special keywords such as `__aligned` and `__saveds`
- ☐ use `#pragma` statements
- ☐ declare and refer to structure members in special ways
- ☐ change the size of enumerated types
- ☐ use the `sizeof` and comma (,) operators in `#if` directives
- ☐ use C++ style and nested comments
- ☐ use national characters in variable names
- ☐ use the common model or the strict reference-definition model for external data.

Using Special Keywords

The SAS/C Compiler allows the following keywords:

- __aligned**
forces an item to be aligned on a four-byte (longword) boundary or forces the declared function to align its stack in the prolog.
- __chip**
places the declared **static** or **extern** data into chip memory.
- __far**
places the declared **static** or **extern** data into the far data section or forces calls to the declared function to be made with 32-bit references.
- __near**
places the declared **static** or **extern** data into the near data section or forces calls to the declared function to be made with 16-bit references.
- __interrupt**
indicates that a function may be called as an interrupt routine.
- __asm**
allows you to specify which registers a function uses to receive parameters.
- __regargs**
forces a function to receive parameters in registers.
- __stdargs**
forces a function to receive parameters on the stack.
- __saveds**
loads the global data register at the entry to the function. The method used depends on the compiler options you specify.
- __inline**
indicates a function that can be inlined by the global optimizer.
- __stackext**
indicates a function that should allocate a new stack if the old one is too small.

The following sections describe each of these keywords. The

- __stackext** keyword is described under “Managing Stack Space.”

When used on functions, these keywords must be specified in the correct places, depending on whether the keyword affects the calling function (*the caller*) or the function being called (*the callee*). The following list summarizes these points.

- Keywords that affect the function's definition (the callee) only may appear on prototypes, but they are optional. These keywords generate warnings if placed on function pointers or data items because they are unnecessary, but the compiler still generates correct code. Keywords in this category are `__aligned`, `__interrupt`, `__saveds`, and `__stackext`.
- Keywords that affect both the caller and the callee must appear both on the function definition and its prototype. You are required to have a prototype for the function and to add the keywords to the declaration of any function pointers. If you do not do this, incorrect code is generated to pass parameters. For example, if you assign a `__regargs` function to a function pointer and compile with the `parms=stack` option, parameters are passed to the `__regargs` function on the stack instead of in registers. Keywords in this category are `__asm`, `__stdargs`, and `__regargs`.
- Keywords that affect only the calling function are required on function prototypes only. The keywords are ignored if they appear on function definitions or function pointers. Keywords in this category are `__near` and `__far`.

Be careful when specifying a keyword on a function pointer. The indirection operator (*) in a function definition resets all special keywords. To declare a pointer to a function that, for example, takes its arguments in registers, declare the pointer as follows:

```
void (* __regargs func)(int x, char *p);
```

If you declare the pointer as in the following example, the `__regargs` keyword is attached to the function pointer itself rather than the function to which the pointer points:

```
void __regargs (*func)(int x, char *p);
```

Using `__aligned`

The `__aligned` keyword on a declaration forces the variable being declared to be aligned on a four-byte data boundary relative to the start of the data area in which it is being declared. For structure members, the data area is the start of the structure. For automatic variables, the data area is the beginning of the stack frame for the function in which they are declared. For `static` and `extern` variables, the data area is the base of the chip, near, or far data section, depending on the section in which the variable is being allocated.

The `__aligned` keyword is valid on both functions and data items. For example, the following code declares the structure `fib` and the function `func`:

```
__aligned struct
    FileInfoBlock fib;

int __aligned func(int x)
{
    /* your code here */
}
```

The SAS/C Compiler normally makes sure that the stack is aligned at the beginning of your program and remains aligned for the duration of the program. However, if your code is called from code not compiled with the SAS/C Compiler, you cannot be sure that the stack is correctly aligned at the start of your program. In that case, the `__aligned` keyword may not work for automatic variables. For example, your code may be:

- ☐ called from assembly-language code
- ☐ an interrupt routine
- ☐ a callback function
- ☐ in a shared library.

To make sure that the stack is aligned at the beginning of a function, declare the function itself with the `__aligned` keyword. This action forces the code generated for the function to align the stack on entry. However, the compiler devotes an address register to pointing to the old stack, thereby reducing the number of address registers available for register variables.

Using `__chip`, `__far`, and `__near`

The `__near`, `__far`, and `__chip` keywords on data items allow you to specify the section in which you want the compiler to place the item. On functions, the `__near` and `__far` keywords are required on the function prototype only.

By default, the compiler puts all declared static or external data into the near data section. Register A4 points to the start of this section. The compiler generates 16-bit references relative to A4 for external or static items in the near data section. These references generate less code than full 32-bit references.

For all items in the far data section, the compiler generates 32-bit references. When the system loader loads your program, it changes these 32-bit references to the actual address of the data item.

The compiler also generates 32-bit references to items placed in chip memory. Chip memory is the lowest 512K to 2M of system memory, depending on the version of the hardware that you are running. This memory is the memory on your machine that is usable by the custom graphics and sound chips to store bitmaps, sound samples, and so on.

You can use the `data` compiler option to change the default data section for external and static data. You can also override the default for individual data items by declaring the item with the `__near`, `__far`, or `__chip` keyword, as in the following example:

```
__near char a[10]; /* Allocate a 10-byte array in the near sec*/
__far  char b[10]; /* Allocate a 10-byte array in the far sec */
__chip char c[10]; /* Allocate a 10-byte array in the chip sec*/
```

For compatibility with previous releases, the compiler accepts the `near`, `far`, and `chip` keywords without the leading underscores. However, these forms of the keywords violate the ANSI Standard and are not allowed if you specify the `ansi` compiler option.

On function declarations, the `__chip` keyword is meaningless. However, functions declared with the `__near` keyword are always called with a 16-bit relative branch even if you compile with the `code=far` option. Functions declared with the `__far` keyword are always called with a 32-bit branch even if you compile with the `code=near` option. For example, the following function is called with a 32-bit branch:

```
__far int func(int x);
```

Using __interrupt

You can declare a function with the `__interrupt` keyword to indicate that the function is called from an interrupt routine. Defining a function with the `__interrupt` keyword turns off the stack checking and extension code for that function and sets condition codes on return. The `__interrupt` keyword is required on function definitions and tolerated on function prototypes. This keyword is meaningless on function pointers and other data items.

Using __asm, __regargs, And __stdargs

Normally, C functions pass and receive parameters by placing them on the stack. To force a function to receive some or all parameters in registers, you can declare the function with the `__asm` or `__regargs` keyword or compile the function with the `parameters=register` option.

If you declare a function with the `__asm` keyword, you can specify the registers in which the function receives each parameter. For more

information about the `__asm` keyword, refer to Chapter 11, “Using Assembly Language with the SAS/C Language,” in *SAS/C Development System User’s Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0*.

If you declare a function with the `__regargs` keyword or if you compile the function with `parameters=register`, the function receives some of its arguments in registers instead of on the stack. Specifically, the first two pointer arguments are in registers A0 and A1, and the first two integral arguments are in D0 and D1. If you compiled with the `math=68881` option, the first two `doubles` are in registers FP0 and FP1. All other arguments are passed on the stack.

The compiler identifies functions that expect arguments in registers to the linker by placing an at sign (@) in front of the function name. The at sign replaces the underscore that the compiler normally places at the beginning of function names.

If you compile your application using `parameters=register` but you want one or more functions to receive parameters on the stack, declare those functions with the `__stdargs` keyword. The `__stdargs` keyword forces a function to receive parameters on the stack.

If you write a function that has the same name as a library function, you need to add the `__regargs` keyword to your function or compile with the `parms=both` or `parms=register` option. For example, you may want to replace the SAS/C library function `malloc` with your own version of `malloc`. If you compile with the `parms=stack` option or define your version of `malloc` with the `__stdargs` keyword, then two versions of `malloc` are linked into your executable. If you use other SAS/C library functions that call `malloc`, these functions use the version of `malloc` in the SAS/C libraries. However, your functions that call `malloc` use your version of `malloc`. To make sure that all calls to `malloc` are using your version of `malloc`, define your version with `__regargs` or compile with the `parms=both` or `parms=register` option.

The `__asm`, `__regargs`, and `__stdargs` keywords are important on function definitions and prototypes, and function pointers.

Using `__saveds` If you define a function with the `__saveds` keyword, the compiler generates extra code at the beginning of the function that loads the address of the near data section into register A4. Defining a function with the `__saveds` keyword is equivalent to compiling that function with the `saveds` compiler option.

Loading A4 with the address of the near data section is useful if the function in question is called from outside your program. For example, your function may be a callback function or may be a function in a shared library.

If you do not compile your function with the `libcode` option, register A4 is initialized with the contents of the absolute symbol `__LinkerDB`, as follows:

```
LEA    __LinkerDB,A4
```

If you compile your program with the `libcode` option, the linker treats `__LinkerDB` as an offset relative to register A6 (the library base) instead of an absolute symbol. Register A6 points to the library base at all entry points to the library. In this case, the instruction generated to load A4 is as follows:

```
LEA    __LinkerDB(A6),A4
```

`__LinkerDB` is initialized by the linker to point to the near data section.

Programs that are residentable cannot refer to absolute symbols. Therefore, you cannot use the `__saveds` keyword or `saveds` compiler option if you intend to link your program with the `cres.o` or `catchres.o` startup modules.

Defining your function with the `__saveds` keyword is equivalent to calling the function `geta4` as the first executable statement of your function. `geta4` is provided for compatibility with previous versions of the SAS/C Compiler and with other Amiga compilers.

The `__saveds` keyword is important in function definitions and is tolerated but not required in function prototypes. If you include the `__saveds` keyword on the prototype for a function, an error message is issued if the keyword is missing from the function definition. This keyword is meaningless on function pointers and other data items.

Using `__inline` If you define a function with the `__inline` keyword, the global optimizer generates code for the specified function at the point the function is called, instead of generating an actual call to the function. Using the `__inline` keyword can increase code size, but it can make your code run faster. It also allows the global optimizer to perform more optimizations. If you do not compile with the `optimize` option, the `__inline` keyword is ignored, and code is generated to call the function normally.

The `__inline` keyword is required on both function prototypes and function definitions. This keyword is meaningless on function pointers.

Managing Stack Space

The *stack* is a writeable memory area whose size is set by the AmigaDOS **stack** command or by the external long integer `__stack`. The default stack size is four kilobytes. This stack is used during function calls for saving registers and passing arguments. Within a function, automatic variables are allocated from the stack. For many C programs, a four-kilobyte stack is adequate.

If you compile your program with the **stackcheck** option (the default) and your program overflows its assigned stack area, the stack overflow routine `__CXOVF` is called. The `__CXOVF` routine provided with the SAS/C libraries displays a requester and aborts the program. However, you can replace the version of `__CXOVF` supplied by the SAS/C libraries with your own version.

If you do not want your program to abort because of a lack of stack space, you can allocate additional stack space in one of four ways:

- Increase the value of the external long integer `__stack`. `__stack` specifies the minimum stack size needed for your program to run. If the default stack is smaller than `__stack`, the startup code allocates a new stack of `__stack` bytes. If you do not compile with **stackext**, the stack space is allocated once by the startup code, so your program does not take longer to run. Changes made to `__stack` after your program starts take effect only if new stack extents are allocated because of code compiled with the `__stackext` keyword or the **stackext** compiler option.
- Declare functions that require a large amount of stack space with the `__stackext` keyword. This keyword generates extra code at the start of the function. This extra code compares the amount of stack available with the amount specified in the external long integer `__STKNEED`. `__STKNEED` specifies the minimum amount of stack needed by each function in your program. If enough stack is not available, the function allocates a new stack extent whose size is specified in the external long integer `__stack`. If there is not enough memory to allocate the new stack, the stack overflow routine `__CXOVF` is called, just as if normal stack-checking code was active instead of the stack extension code.

When the function returns, the old stack is restored and the extra stack is freed. Declaring a function with the `__stackext` keyword is equivalent to compiling the function with the **stackext** compiler option.
- Compile your program with the **stackext** option. Compiling with the **stackext** option is equivalent to defining all functions in your program with the `__stackext` keyword. If you have specific functions that use a lot of stack space or are recursive, you may want

to define those functions with the `__stackext` keyword instead of compiling your entire program with the `stackext` option.

- Increase the value of the external long integer `__STKNEED` and compile your program with the `stackext` option. You should never set `__STKNEED` to less than 400 bytes (which is the default value).

Note: `__stack` specifies the total number of bytes needed during the entire duration of your program. `__STKNEED` gives the minimum free stack required by each function.

Note: Using the stack extension feature adds overhead to each function entry point and requires the use of an additional address register. Therefore, your program will run slower.

The `__stackext` keyword is important only on function definitions. This keyword is meaningless on function pointers and tolerated on function prototypes, but not required. If you include the `__stackext` keyword on the prototype for a function, an error message is issued if the keyword is missing from the function definition.

Even if you compile your program with `stackext`, your program may still run out of stack space if it calls a function not compiled with `stackext`. If you call shared libraries or code compiled with other languages, your program may run out of stack space. Also, the SAS/C library routines are not compiled with `stackext`, but they use little stack space. The stack extension code allows some extra space for the library routines.

In the same application, you can use functions compiled with `stackcheck`, functions compiled with `stackext`, functions compiled with neither option, and functions defined with the `__stackext` keyword. If you compile a function with both `stackext` and `stackcheck`, the compiler ignores the `stackcheck` option.

For more information on the `__stack` integer, refer to *SAS/C Development System Library Reference, Version 6.0*.

Using #pragma Statements

The SAS/C Development System supports five types of `#pragma` statements:

- `flibcall`
- `libcall`
- `syscall`
- `tagcall`
- `msg`.

The `libcall`, `syscall`, and `tagcall` `#pragma` statements allow you to call various library routines directly. These `#pragma` statements are

described in *SAS/C Development System Library Reference*. The **flibcall #pragma** statement also allows you to call various library routines directly. The **flibcall** statement is similar to **libcall**, except that each byte instead of each nibble is interpreted as a register. **#pragma flibcall** allows you to use floating-point registers. However, you can only pass 6 parameters instead of fourteen. Currently, the AmigaDOS system libraries do not accept parameters in floating-point registers. However, if you create your own libraries that accept parameters in floating-point registers, you should use **#pragma flibcall**.

The SAS/C Compiler also supports **#pragma msg** statements. **#pragma msg** statements allow you to specify whether a message should be ignored, treated as an error, or treated as a warning message. For example, if you know that your code produces a specified warning on a certain line, you can use a **#pragma msg** statement to suppress the warning on that line.

Note: You cannot turn a message that is by default an error into a warning.

How a specific message is treated is referred to as its *state*. Using **#pragma msg** statements, you can save the state of a message, change its state, or restore a message's previous state.

The **msg** statement can take one of the following forms:

```
#pragma msg num [error | warn | ignore] [push]
```

```
#pragma msg num pop
```

where

- num** identifies the number of the message.
- error** indicates that the message should be treated as an error. You can abbreviate this keyword as **err**.
- warn** indicates that the message should be treated as a warning, if possible. You can abbreviate this keyword as **wrn**.
- ignore** indicates that the message should be ignored, if possible. You can abbreviate this keyword as **ign**.
- push** saves the message's state (before changing it). If you specify **push**, this keyword must appear after the **error**, **warn**, or **ignore** keyword.
- pop** restores the message's previous state.

With the **push** and **pop** keywords, you can turn a message on or off for a specific line, series of lines, or header file, then restore the message to the value the user specified on the command line.

For example, the following code produces message number 85 because it returns an `int` but no return value is specified. However, the code is correct because the `exit` function never returns. The first `#pragma` statement tells the compiler not to produce a warning on that line. The second `#pragma` tells the compiler to generate a message for the remaining functions in the file, if necessary.

```
int func(void)
{
    .
    .
    .
    exit(0); /* Never returns */
#pragma msg 85 ignore /* Turn off warning 85 */
}
#pragma msg 85 warning /* Turn on warning 85 */

int func2(void)
{
    .
    .
    .
} /* A warning here is VALID and is produced due to the */
/* second #pragma statement above. */
```

Using Unnamed Unions

The SAS/C Compiler supports unnamed unions as structure members. For example, you could define the following structure:

```
struct F00
{
    union
    {
        int x;
        double d;
    };
    long l;
} foo;
```

The union in this structure does not have a name. The union members are treated as if they are members of the structure, except that the union

members start at the same offset relative to the base of the structure. In this example, you can refer to member `x` as `foo.x`.

Using Implicit Structure References

If you declare a substructure in your program, and the names of the members in the substructure are unique, you can refer to the members using only the structure name and member name. You do not need to include the substructure name in the reference. For example, you could declare the structure `BAR` and the substructure `FOO` as follows:

```
struct FOO
{
    int x;
    double d;
};

struct BAR
{
    struct FOO foo;
    int l;
} bar;
```

You can refer to members `x` and `d` in the substructure as follows:

```
bar.x = 10;
bar.d = 10.0;
```

Referring to substructure members as shown generates warning message 193. If your program refers to substructure members in this way, suppress warning 193 by compiling your program with the `ignore 193` compiler option.

If two or more substructures contain a member with the same name and you do not specify the substructure name when you refer to the member, the compiler generates error message 123. The compiler cannot determine which substructure member you are trying to reference.

Using Equivalent Structures

If you compile your program with the `StructureEquivalence` option, the compiler does not issue messages if a pointer to one structure type is passed to a function when the function expects a pointer to a different type, if the type passed is equivalent to the type expected.

A structure is defined to be *equivalent* if it maps the same base data types in the same order as the desired structure type. If one structure is longer than another, but it is identical through the length of the shorter one, then the longer structure is an acceptable substitute for the shorter, but the shorter structure is not an acceptable substitute for the longer structure. The function receiving the pointer may try to write to one of the fields that does not exist in the shorter structure. Therefore, you can pass an `IntuiMessage` structure to the exec function `PutMsg` without getting a warning, but the compiler issues a warning if you assign the result of a call to `GetMsg` to an `IntuiMessage` pointer.

For example, you could define the following structures:

```
struct FOO
{
    int x;
    int y;
};

struct BAR
{
    int x, y, z;
};
```

The structure `BAR` is equivalent to the structure `FOO`. If you specify **structureequivalence**, the compiler does not issue a warning for passing a pointer of type `BAR` to a function whose prototype specifies a structure of type `FOO`. However, the structure `FOO` is not equivalent to `BAR` because `FOO` is not long enough. If you tried to pass a pointer to a structure of type `FOO` to a function whose prototype specified a pointer to a structure of type `BAR`, you would get a compiler warning.

`BAR` is also equivalent to `FOO` if you define them as follows:

```
struct FOO
{
    int x, y;
};

struct BAR
{
    struct FOO foo;
    int z;
};
```


Similarly, you can pass a `struct IntuiMessage *` to a function that requires a `struct Message *`, since a `struct Message` is the first element of a `struct IntuiMessage`. (`struct IntuiMessage` is defined in the system header file `intuition/intuition.h`.) This option also allows you to pass a pointer to an `IOExtSer` structure to a routine that wants a pointer to an `IoRequest` structure, such as the routine `DoIO`. `IOExtSer` is identical to `IoRequest` for the length of the `IoRequest` structure.

The following example is more complicated, but `BAR` is still equivalent to `FOO`:

```
struct FOO
{
    int x, y, z;
};

struct BAR
{
    struct
    {
        int a;
    } x;
    struct
    {
        int b, c;
    } y;
    int z;
};
```

The structure type `FOO` defines memory as three consecutive integers. `BAR` also defines memory as three consecutive integers, although it does so through the use of two different substructures. The substructures are not considered when determining equivalence.

Using Zero-Length Arrays

You can use zero-length arrays as the last element of a structure. For example, you can declare structure `FOO` as follows:

```
struct FOO
{
    long maxalloc;
    long curalloc;
```

```
    long mem[0];
};
```

This structure is 8 bytes in length as reported by `sizeof`. Using zero-length arrays is most useful when you need to allocate variable-sized data items. For example, if you need 100 elements in the above array at run-time, you can do the following:

```
struct FOO *foo;
foo = malloc(sizeof(struct FOO) + 100*sizeof(long));
foo->mem[50] = 10;
foo->mem[56] = ....
```

Specifying the Size of Enumerated Types

By default, enumerated types are integers and integers are 4 bytes long. For example, each of the identifiers of type `cats` is 4 bytes long.

```
enum cats {bombay, ocicat, somali};
```

If you compile this line with the `shortintegers` option, the identifiers are 2 bytes long.

The SAS/C Compiler also allows you to control the size of enumerated types by declaring them with the `char`, `short`, or `long` keywords, as in the following examples:

```
char enum flintstones {Fred, Barney, Wilma};
short enum colors {red, green, blue, black, white};
long enum seasons {spring, summer, fall, winter};
```

The identifiers of type `flintstones` are 1 byte long; the `colors` are 2 bytes long, and the `seasons` are 4 bytes long. When you declare a variable of a specific `enum` type, you must include the extra keyword on the variable's declaration:

```
char enum flintstones var1;
```

Using the sizeof and Comma Operators in Preprocessor Directives

The SAS/C Compiler allows you to use the `sizeof` and comma `(,)` operators in preprocessor `#if` directives, as in the following example:

```
#if (sizeof(int) == 2)
    #error short integers
#endif
```

However, the ANSI Standard does not allow these operators in preprocessor directives. Therefore, the compiler produces a warning message if you compile with your program with the `ansi` or `strict` option.

Using C++ Style and Nested Comments

In C++ programs, two consecutive slashes `(//)` define a comment block that extends to the end of the current line. You can use two slashes to designate comments in your SAS/C programs. However, if you compile with the `ansi` or `strict` options, the compiler generates a warning message for these comments.

The ANSI Standard states that a C program cannot have nested comments. Therefore, if you enter the comment start sequence `(/*)` inside of a comment, the sequence is not recognized as the beginning of a second, or nested, comment. The first comment end sequence `(*/)` terminates comment mode. However, if you specify the `commentnest` compiler option, as described in Chapter 8, “Compiling and Linking Your Program,” the compiler allows nested comments.

Using National Characters in Variable Names

If you do not compile your program with the `ansi` option, you can use accented characters in variable names.

Using Common Model External Data

The ANSI Standard does not define how external data should be handled. Typical UNIX compilers allow you to declare external variables in multiple places, as long as you do not initialize the variable in more

than one of those declarations. For example, you could define the following variable in a header file:

```
int i;
```

If this header file is included by all files in a project, each file defines a variable `i`. The linker merges all these independent definitions into one, creating a *common model* for external data.

Most microcomputer compilers use what is called the *strict reference-definition model* (or *strict ref-def*) for external data. This model requires that one and only one source file in a project define the external variable. The remaining files in a project can only declare the external variable. To change the definition of variable `i` to a declaration, use the `extern` keyword as follows:

```
extern int i;
```

The SAS/C Compiler supports both the common model and the strict reference-definition model. The default model is strict reference-definition. If you want to use the common model, compile your program with the `common` compiler option.

12 How Does the Compiler Work?

171	<i>Introduction</i>
171	<i>Compiling Your Program</i>
173	<i>Linking Your Program</i>
174	<i>Running Your Program</i>
174	<i>Stack Area</i>
175	<i>Heap Area</i>
175	<i>Changing Hunk Names</i>
176	<i>Addressing Data</i>
177	<i>Understanding Data Types and Sizes</i>
179	<i>Storing Data</i>

Introduction

Programming with the SAS/C Compiler is not much different than programming in C on other compilers. Most of the books about the C language that are at your local bookstore are applicable to the SAS/C Compiler. If you confine your programming to the features that these books describe, you do not need much specific information about how the SAS/C Compiler works. However, if you want to perform special tasks such as linking with assembly language functions, writing I/O drivers in C, or using overlays, you may need to know how your C program is translated into executable code.

This chapter describes how your program is translated into executable code. This chapter assumes a basic knowledge of object file format as described in *The AmigaDOS Manual, 3rd Edition* (Commodore-Amiga, Inc. 1991).

Compiling Your Program

The compiler produces an object module (a .o file) for each assembler or C source file. Each object module is organized into hunks. In general, a hunk contains the following information:

- program code or data. The program code is written into the code hunk. The data for your program are written into one of several hunks, depending on how the data are declared. For example, data declared with the `__chip` keyword are placed in the chip hunk.

- external symbol information, including subroutine entry points and global data. This information is also referred to as global symbols or **XDEFS**.
- relocation records. The linker uses these records to produce a load file. A load file is also called an *executable module*.

Your program code and data are organized into the following hunks:

Code hunk

contains the machine instructions for your program, any string constants (if you compile with **stringmerge**), and anything else that the compiler decides is definitely read-only. By default, this hunk is named **text**, but you change this name with the **codename** option. See the section “Changing Hunk Names,” later in this chapter, for more information.

Near data hunk

contains all initialized near data, including constants and string constants (if you do not compile with the **stringmerge** option). *Near data* are data that are declared with the **__near** keyword or are defined in modules compiled with the **data=near** option. **data=near** is the default setting, so unless you specify otherwise, all of your initialized data are placed in the near data hunk. Near data can be accessed by resident programs and libraries. This hunk is named **__MERGED**.

Near BSS hunk

specifies the amount of memory to allocate for all uninitialized near data. The startup code initializes the near BSS hunk to 0. This hunk is also named **__MERGED**.

Chip hunk

contains all data that are declared with the **__chip** keyword or compiled with the **datamem=chip** option. This hunk is named **chip**.

Far data hunk

contains all initialized far data. *Far data* are data that are declared with the **__far** keyword or are defined in modules compiled with the **data=far** or **data=faronly** options. By default, this hunk is named **data**, but you can change the name with the **dataname** option. See the section “Changing Hunk Names,” later in this chapter, for more information.

Far BSS hunk

specifies the amount of memory to allocate for all uninitialized far data. The system loader allocates this memory and sets it to zero. By default, this hunk is named **udata**, but you can change the name with

the **bssname** option. See the section “Changing Hunk Names,” later in this chapter, for more information.

When you compile your program with the **verbose** option, the compiler produces a message in the following format:

```
Module size P=00000000 D=00000000 U=00000000 C=00000000
F=00000000 UF=00000000
```

The letters indicate which hunks were produced for your program:

- P code hunk (your program)
- D near data hunk
- U near BSS (uninitialized data) hunk
- C chip hunk
- F far data hunk
- UF far BSS (uninitialized data) hunk

The numbers give the size in bytes, using hexadecimal notation, of each hunk. If the compiler does not create certain hunks, then the compiler either displays zeroes or does not display any numbers for these hunks.

If you compile your program using a **debug** option, the compiler generates debugging information and writes this information into two types of hunks that hold debugging information:

- H_DEBUG** contains line numbers, and information about variables, **typedefs**, and structures. **H_DEBUG** hunks are generated by the compiler or the assembler.
- H_SYMBOL** contains absolute addresses of global symbols. **H_SYMBOL** hunks are generated by the compiler or the linker.

Linking Your Program

The linker combines the object modules to produce a single executable module or shared library. To produce an executable module, the linker does the following:

- ☐ reads each of the object modules.
- ☐ reads each of the link libraries, if necessary.
- ☐ writes the contents of the object modules to the executable module. In the process, the linker resolves all intra-hunk and near data relocations

and generates relocation records for relocations that span hunks. The linker also combines any hunks in the object module that have the same name. For example, the linker merges all near BSS hunks and appends them to the `__MERGED` data hunk.

- ☐ generates overlay nodes, if required.
- ☐ copies segments of the library files that are referenced by the program to the executable module. If those segments reference any symbols, then the library segments that define those symbols are also copied to the executable module. This process is repeated until all symbols are resolved.
- ☐ produces data for the overlay supervisor, if required.
- ☐ produces a map file and cross-reference tables, if requested.

If you are not using overlays, all of the hunks in the executable module are contained in one *primary node* (also called the *root node*). If you are using overlays, the executable module also contains a node for each overlay.

Running Your Program

When you run your program, the AmigaDOS system loader copies the executable module into memory. As it copies the executable module, the loader:

- ☐ resolves the remaining relocations.
- ☐ allocates and zeroes the memory required for uninitialized far data (the far BSS hunks).

There are two other program components, the stack and the heap, that are not part of the executable file but that form an important part of the final executing program.

Stack Area The *stack* is a writeable memory area whose size is set by the AmigaDOS `stack` command or by the external long integer `__stack`. The default stack size is four kilobytes. This stack is used during function calls for saving registers and passing arguments. Within a function, automatic variables are allocated from the stack.

For many C programs, a four-kilobyte stack is adequate. If your program requires more stack space, see Chapter 11, “Using Amiga Specific Features of the SAS/C Language,” for information on allocating additional stack space.

Heap Area The *heap* is a writeable memory area whose size is determined by the following:

- ☐ the dynamic memory needs of the program
- ☐ the amount of memory installed on the system
- ☐ the amount of memory that is currently being used.

All memory allocation routines such as `malloc` return memory from the heap. If the heap is not large enough to handle a request, the library function calls on the AmigaDOS system to provide more memory.

Changing Hunk Names

If you change the names of any of the hunks produced by the compiler, keep in mind the following:

- ☐ All hunks with the same name are merged by the linker.
- ☐ Hunks with different names are loaded separately by the system loader.
- ☐ The system loader may have trouble finding enough contiguous memory to load very large hunks.

The following hunk section names are reserved:

NTRYHUNK

Your load file may contain only one **NTRYHUNK** hunk. The linker places this hunk first in the executable module. Usually, this name is used only as the name of overlay manager's code hunk.

—MERGED

This name is used for the near data section. All hunks with this name are merged and moved into the root node. The startup code initializes register A4 to point to this hunk.

—NOMERGE

Hunks with this name are never merged with other hunks, including other hunks named **—NOMERGE**. They occupy a single, separate hunk in the load file.

Addressing Data

If you compile with the default options, your program code and data are referenced with one of three different types of addresses:

- 16-bit address relative to the program counter.

Items in the code section are referenced with 16-bit PC-relative addresses.

The 16-bit offset is a signed value that is added to the current 32-bit address in the program counter. Therefore, if a function is more than 32 kilobytes above or below the point at which it was called, the linker constructs an absolute branch instruction within the 32-kilobyte range and routes the call through that branch.

- 16-bit address relative to register A4.

When you link your program, all data in the near data and near BSS hunks are merged into one near data section. When you run your program, the startup module loads the address of the near data section into register A4, and this section is referenced with 16-bit A4-relative addresses.

Using 16-bit addresses produces code that is smaller and faster because each load or store instruction requires only four bytes. However, because near data are referenced with 16-bit addresses, you are limited to 64 kilobytes of near data. If you have a very large data structure or array, consider placing it into the far section.

- 32-bit absolute address.

Items referenced with 32-bit addresses can be located anywhere in memory. Items in the far data and far BSS sections are referenced with 32-bit absolute addresses.

Using 32-bit addresses produces code that requires more space and takes longer to run because each load or store instruction requires six bytes. However, because far data are referenced with 32-bit addresses, there is no limit on the amount of far data that you can declare in your program.

chip data are always referenced with a 32-bit pointer and are therefore considered non-reentrant. However, most chip items are picture image structures that are referenced in a read-only mode. If your program uses chip data as read-only, your program is still re-entrant. Chip data can be accessed by the Amiga custom chips such as graphics chips and sound chips. Chip data are loaded into chip memory, which is the lowest 512K to 2M of system memory, depending on the version of the hardware that you are running.

You can modify how your program code and data are accessed by:

- declaring individual data items with the appropriate keyword:
`--chip`, `--near`, or `--far`

- compiling with the **stringmerge** option
- compiling with the **absfuncpointer** option
- compiling or linking with the **smallcode** and/or **smalldata** options
- compiling with the **code** and/or **data** options
- compiling with the **datamem**, **codemem**, and/or **bssmem** options
- linking with the **chip** or **fast** option.

For more information on the **__chip**, **__near**, or **__far** keywords, see Chapter 11, “Using Amiga Specific Features of the SAS/C Language.” For more information on the compiler and linker options, see Chapter 8, “Compiling and Linking Your Program.”

Understanding Data Types and Sizes

The following sections describe how various data types are represented on Amiga hardware.

By default, all arithmetic objects are signed, and therefore, they can take on values less than zero. However, if you specify the **unsignedchar** option, the compiler treats all **char** objects as unsigned.

You can specify that an integral object (**char**, **int**, **short** or **long**) be signed or unsigned by declaring the object with the **signed** or **unsigned** keyword, as in the following example:

```
unsigned long x;
```

The following table lists the sizes, minimum values, and maximum values of each of the signed and unsigned data types.

Type	Bytes	Minimum	Maximum
signed char	1	-128	+127
unsigned char	1	0	255
signed short	2	-32,768	+32,767
unsigned short	2	0	65,535
signed int	4	-2,147,483,648	+2,147,483,647
unsigned int	4	0	4,294,967,295

(continued)

Type	Bytes	Minimum	Maximum
signed long	4	-2,147,483,648	+2,147,483,647
unsigned long	4	0	4,294,967,295
float (IEEE)	4	3.402E-37	3.402E+38
double (IEEE)	8	2.222E-308	1.797E+308
float (FFP)	4	5.421E-20	9.223E+18
double (FFP)	4	5.421E-20	9.223E+18

Note: In the AmigaDOS environment, pointers are always 4 bytes in length. Also, the values for **signed int** and **unsigned int** in this table are the values if you do not specify the **shortint** option.

The default size of an **int** on Amiga hardware is four bytes. However, you can set the size of an **int** to two bytes by compiling your program with the **shortint** option.

When the compiler is calculating the value of an expression using variables of different data types, the compiler performs some automatic type promotions and then does the calculation using the data type of the widest operand. The *widest* operand is the operand that can contain the largest positive number. For example, suppose you may have a calculation involving a variable of type **char** and a variable of type **unsigned long**. The **char** is automatically promoted to **int**, so the calculation now involves an **int** and an **unsigned long**. The **int** is then promoted to **unsigned long** under the wider type rule.

These rules can dramatically affect your code. In the example calculation, the **char** is signed. If it holds a **-1**, for example, the **-1** is promoted to **int**, yielding **-1**. The **int** is then promoted to **unsigned long**, yielding a very large positive number.

These implicit conversions follow the conversions described in the ANSI Standard in Section 3.2.1.5, "Usual Arithmetic Conversions." The following table lists the type to which the various data types are converted.

Operand Type	Conversion Type
<code>char</code>	<code>int</code>
<code>unsigned char</code>	<code>int</code>
<code>short</code>	<code>int</code>
<code>unsigned short</code>	<code>int</code>
<code>int</code>	<code>int</code>
<code>unsigned int</code>	<code>unsigned int</code>
<code>long</code>	<code>long</code>
<code>unsigned long</code>	<code>unsigned long</code>
<code>float</code>	<code>float</code>
<code>double</code>	<code>double</code>

Note: As previously stated, if you compile your program with the `shortint` option, integers are two bytes long. Then, if you cast the result of an expression involving two integers to another type, you do not get the results you might expect. For example, you could have the following code in your program:

```
long l;
int i=32000;
int j=32000;

l = (long)(i * j);
```

If you compile this code with `shortint`, `i` and `j` are not promoted to `long` before the expression is evaluated. The result of the expression is too long to fit into a short integer. The overflow is cast to a `long`, which gives the wrong result. To correct the problem, explicitly cast `i` or `j` to a `long`, as in the following example:

```
l = (long)i * j;
```

Storing Data

In the C Language, each data object is associated with a *storage class* that is either declared with a keyword (`auto`, `extern`, `static`, or

register) or determined by the context in which the declaration occurs.

An object's storage class determines the location where the object is stored and the scope of the object. Therefore, the storage class of the objects in your program can affect your program's size and performance.

The following list describes where objects are stored, based on their storage class:

External

External objects are stored in a global data area (one of the near, near BSS, far, far BSS, or chip sections).

Static

Static objects are stored in a global data area (one of the near, near BSS, far, far BSS, or chip sections) unless you specify the **stringmerge** option. If you specify **stringmerge**, data declared **static const** are placed in the code section.

Automatic

Storage for automatic objects is allocated on the stack during the execution of the function in which the object is defined.

Formal

An object has the storage class *formal* if it is a parameter (or argument) to a function. Storage for formal objects is allocated on the stack during the execution of the function in which the object is defined.

Register

Register variables may be stored in registers, or they may be stored on the stack. If you run the global optimizer, the optimizer allocates all register variables and ignores the **register** keyword. If you do not run the optimizer and you specify the **noautoreg** option, only those variables you declare with the **register** keyword are stored in registers. If you compile with **autoreg**, the code generator decides which additional variables are stored in registers.

13 Writing Portable Code

181	<i>Introduction</i>
181	<i>Compiling Your Program</i>
182	<i>Writing Your Program</i>
183	<i>Dealing with Data Type Sizes</i>
183	<i>Determining Structure Size and Padding</i>
184	<i>Writing Structures to a Disk File</i>
187	<i>Using Narrow Types in Pre-ANSI Function Declarations</i>
188	<i>Using Incomplete Structure Tags</i>

Introduction

The C language has many characteristics of a portable language, but some features of the C language are dependent on the type of machine on which the program is running. However, you can write programs that run on many different machines if you follow some guidelines when compiling and writing your program.

The following sections describe actions you can take to improve the portability of your programs.

Compiling Your Program

You can compile your program with the `strict` and/or `ansi` options. These options enable many warnings that identify possible problems in your code that may cause errors if you port your code to systems other than the Amiga system. For example, the following code generates message number 120:

```
void func1(long);

void func2(void)
{
    short s=0;
    func1(s);
}
```


If you compile this code with the `strict` option, the compiler displays the following message:

```
Warning 120: Integral type mismatch: possible portability problem
           Expecting "long", found "short"
```

Writing Your Program

If you use only library functions and language features that are defined in the ANSI Standard (*American National Standard for Information Systems—Programming Language C*), your program should compile without modification on any system that has an ANSI-compliant C compiler. However, there are many features of the C language that are implementation-defined. Even if your program compiles without generating errors, it may not run as you expect. See Appendix 3, “Implementation-Defined Behavior,” for more information.

To help you identify non-portable functions and macros in your program, you can define the `__STRICT_ANSI` preprocessor symbol. You can define this symbol by entering `define __STRICT_ANSI=1` in the `sc` command or by entering the following statement in your C source file before any header files are included:

```
#define __STRICT_ANSI 1
```

If you define `__STRICT_ANSI`, the header files that are defined by the ANSI Standard (such as `errno.h` and `stdio.h`) do not define any prototypes or macros for functions not permitted by the ANSI Standard. If your program calls a non-ANSI function, the compiler issues a warning message because the function has no prototype. For more information on the `__STRICT_ANSI` symbol, refer to *SAS/C Development System Library Reference, Version 6.0*.

In addition, if you follow a few additional guidelines, you can avoid most portability problems. The following list contains some guidelines that can help improve the portability of your program.

- ☐ Do not assume data type sizes are standard.
- ☐ Do not assume structure padding or size is standard.
- ☐ Be careful when writing structures to disk.
- ☐ Do not use narrow types in old-style definitions.
- ☐ Be careful about using incomplete structure tags.
- ☐ Do not write pointers to files. Pointers are useless outside of the invocation of the program that generates them.

- Do not assume that you know whether `char` variables are signed.
- Do not assume that you know the placement of bitfields within structures.

The following sections discuss these guidelines in more detail.

Dealing with Data Type Sizes

Do not assume that the `int` data type is always a particular length. The ANSI Standard specifies that the `int` type must be at least as long as the `short` type and that the `short` data type must be able to hold numbers in the range 32767 to -32768. Therefore, portable code should never store numbers larger than 32767 or smaller than -32768 in an `int`. Use `long` for larger numbers.

The default size of an `int` on the Amiga system is four bytes. On other systems (such as the IBM PC), the size of an `int` is two bytes. The SAS/C Compiler treats the data types `int` and `long` as the same data type. However, you can set the size of an `int` to two bytes by compiling your program with the `shortint` option. If you compile with `shortint`, the compiler treats the data types `int` and `short` as the same data type.

If you port your program to a system that uses two-byte integers and you do not compile with the `shortint` option, you may receive unexpected results. For example, when passing parameters on the Amiga system, `short` and `char` variables are silently promoted to `ints`. Therefore, any function declared as receiving a `long` does not produce a warning if you pass a `short` or `char` to the function. To receive an appropriate warning message, you can enable warning 120 by compiling with the `warn=120` or `strict` compiler options.

Declaring an object as `signed` is usually redundant, because integral objects are signed by default. However, judicious use of the `signed` keyword can enhance your program's portability, because some compilers may make `char` declarations and bitfields `unsigned` by default.

Determining Structure Size and Padding

Even though the Motorola 68000 processor uses byte addresses, it requires that 16-bit and 32-bit data items be aligned on even addresses. That is, the lowest bit of the address must be zero. Because of this requirement, the SAS/C Compiler inserts dummy bytes (called *padding*) as necessary to align integers, `floats`, `doubles`, and pointers.

For example, in the following structure, the SAS/C Compiler adds one byte between the structure members `a` and `l`.

```
struct misc
{
    char a[3];
```

```

        long l;
    };

```

If the dummy byte was not added, the `long` structure member would end up on an odd boundary and an addressing exception would result.

On other systems, an ANSI-compliant C compiler may add more or less padding as required. Many computers require that long integers fall on four-byte boundaries, so the compiler would add three bytes of padding instead of one.

Because of these alignment requirements, data structures may be different sizes on different machines. Do not assume that you know the size of a structure based on the fields declared in it. Use the `sizeof` operator to determine the size, and do not depend on `sizeof` to return the same result on different machines. Using the previous example, the expression `sizeof(struct misc)` has a value of 8, and the following expression has a value of 7:

```
sizeof(misc.a) + sizeof(misc.l)
```

Writing Structures to a Disk File

If you try to write structures to a disk file on one system and read the file on another system, you may get different values. This problem may be caused by differences in structure padding (as described under “Determining Structure Size and Padding”) or by differences in the representations for characters, floating-point numbers, or integers. For example, the following program writes a structure to a file:

```

#include <stdio.h>
main()
{
    FILE *fp;
    struct
    {
        char x;
        short y;
    } record;

    fp = fopen("testfile","wb");
    record.x = 3;
    record.y = 4;
    fwrite(&record,sizeof(record),1,fp);
    fclose(fp);
}

```

If you compile and run this program using the default compiler options, the file `testfile` contains four bytes in the following order: `03 00 00 04`. However, if you compile and run this program under MS-DOS, which uses the Intel 8086 processor, the file contains `03 04 00`. The reasons for the different values are as follows:

- The SAS/C Compiler inserts a padding byte so that integer `y` is correctly aligned. The Intel processor does not require this padding.
- The Motorola processor writes integers from the high byte to the low byte, and the Intel processor writes integers from the low byte to the high byte.

The following paragraphs describe some simple rules that you can use to make your structures more portable between the Amiga computer and most other computers. These guidelines help you produce a structure that your program can write to disk and then read in a single operation. These guidelines are not guaranteed to work in all cases, but they do gain you limited ability to interchange files between various computers.

The computer to which you are porting your program must do the following:

- Use the same floating-point format for doubles that you are using on the Amiga system (FFP if the `math=ffp` option is specified, IEEE for all other formats).
- Use the ASCII character set.
- Use the same sizes for data types:

<code>char</code>	1
<code>short</code>	2
<code>long</code>	4
- Store integers using the same bit pattern that the Amiga system does.

The structure to be written must follow these rules:

- The structure can contain only the following data types:

Type	Length
<code>char</code>	1
<code>short</code>	2
<code>float</code>	4
<code>long</code>	4
<code>double</code>	8 (or 4 if you compile with <code>math=ffp</code>)

Do not use `ints` or any pointer types in the structure.

- Structures or unions may contain other structures or unions that follow these rules.
- Each field must be placed at an offset relative to the start of the structure that is an exact multiple of the field's size.

- A nested structure or union must be placed at an offset that is an exact multiple of the largest simple arithmetic type occurring in any of its fields, including the fields of any substructures or subunions.
- If the structure or any of its substructures contains a `double`, the structure must contain extra fields as needed to ensure that its total size is an exact multiple of the size of a `double`. If it does not contain a `double`, it must contain extra fields to ensure that its total size is an exact multiple of four.

For example, you cannot port the following structure between systems:

```
struct NOTDISKABLE /* Non-diskable structure */
{
    char a;
    double d;
    short s;
    long l;
    char name[27];
    struct XXX
    {
        char a, b;
    } foo[11];
};
```

To make the previous structure portable between systems, you must declare the structure as follows:

```
struct DISKABLE /* Diskable version of same structure */
{
    char a;
    char dummy1[7]; /* following double needs 8-byte offset */
    double d;
    short s;
    short dummy2; /* following long needs 4-byte offset */
    long l;
    char name[27];
    char dummy2[5]; /* following structure needs 4-byte offset */
    struct XXX
    {
        char a, b;
        char dummy3[2]; /* Pad to multiple of 4 bytes */
    } foo[11];
    char dummy4[4]; /* Pad to multiple of 8 bytes */
};
```

Using Narrow Types in Pre-ANSI Function Declarations

The ANSI Standard states that function definitions should specify the data type of each parameter, as in the following example:

```
/* ANSI Standard prototype-style definition */
void newfunc(short s, float f)
{
    .
    .
    .
}
```

Pre-ANSI C accepted function definitions in the following form:

```
/* Pre-ANSI old-style definition */
void oldfunc(s, f)
short s;
float f;
{
    .
    .
    .
}
```

The ANSI Standard default argument promotion rules (described in Section 3.3.2.2 of the ANSI Standard) differ for functions defined with prototype-style definitions and functions defined with old-style definitions. You may get incorrect results on ANSI-conforming compilers if you use ANSI standard prototypes and old-style function definitions for the same function. You can avoid this problem by not using narrow data types (**char**, **short**, and **float**) in function prototypes and definitions.

The ANSI Standard says that functions defined with old-style definitions and no prototypes must receive all **char** and **short** parameters as if they were type **int** and receive all **float** parameters as type **double**. The incoming parameters are then converted to the appropriate shorter type. Functions defined with new-style definitions are free to receive narrow types as declared. Unfortunately, the caller of the function does not know whether the function is defined with a prototype-style definition or with an old-style definition.

The ANSI Standard does not define what happens if you declare a function using an old-style definition but also include an ANSI standard prototype for the function. To make more programs work as expected, the SAS/C Compiler treats such definitions as if they were new-style definitions.

If you do not port your program to a system other than the Amiga system, you should not encounter any problems with narrow types unless the following situation occurs:

The caller has a prototype for a function that says `float`. The called function does not include the prototype and has an old-style definition that says `float`.

The data types `char` and `short` do not cause problems on the Amiga system because they are passed as if they were type `int` for both old-style and prototype-style function definitions.

To help identify problem situations, you can enable the following warnings:

Warning	Description
165	This message identifies narrow types that have been used in function prototypes. You can enable this warning with the <code>warn=165</code> option.
176	This message identifies arguments that have been promoted using the default argument promotion rules and, as a result, conflict with the prototype given for the function. You can enable this warning with the <code>strict, ansi, or warn=176</code> option.
179	This message identifies narrow types that have been used in an old-style function definition. You can enable this warning with the <code>strict or warn=179</code> option.

Using Incomplete Structure Tags

An *incomplete structure tag* is a structure tag (or *name*) that has not yet been defined. The ANSI Standard allows you to refer to structures by name before they have been defined, as long as you only declare pointers to the structure and do not attempt to dereference these pointers.

Incomplete structure tags can cause problems if they occur in a function prototype. The ANSI Standard allows incomplete structure tags in function prototypes but states that the incomplete tag goes out of scope at the closing parenthesis of the prototype. Any declaration of the structure later in the program is not associated with the incomplete tag used in the prototype and is considered to be a completely different structure.

If you define the function later in the program, an ANSI-conforming compiler may issue an error message, and your program may not compile.

For example, many ANSI-compliant compilers generate an error message for the following code, claiming that the type of the parameter does not match the type declared in the prototype.

```
void func(struct FOO *); /* Incomplete tag FOO */

struct FOO
{
    int x, y, z;
};

void func(struct FOO *parm) /* Error here! */
{
}
```

If you move the definition of `struct FOO` before the prototype, this code compiles without problems.

To help you identify incomplete tags, you can enable the following warnings:

Warning	Options
148	This warning identifies any use of incomplete tags. You can enable this warning with the <code>warn=148</code> option.
149	This warning identifies incomplete tags that are used in function prototypes. You can enable this warning with the <code>strict</code> or <code>warn=149</code> option.

■ Part 4

Appendices

Appendices	1	Solving Common Problems
	2	Error and Warning Messages
	3	Implementation-Defined Behavior
	4	Converting From Aztec C Options to SAS/C® Options
	5	Converting from Version 5 to Version 6

Appendix 1

Solving Common Problems

193	<i>Introduction</i>
193	<i>Resolving Undefined Symbols</i>
195	<i>Fixing Compilation Errors</i>
197	<i>Using CodeProbe</i>
197	<i>Using Formatted Print Functions</i>
198	<i>Using getchar</i>
199	<i>Getting Incorrect Results from Function Calls</i>
200	<i>Crashing the Machine</i>
200	<i>Using The asctime Function</i>
201	<i>Managing the Standard I/O Window</i>
201	<i>Linking Resident Programs or Creating Resident Libraries</i>
202	<i>Writing Replacement Functions for SAS/C Library Functions</i>
202	<i>Eliminating Informational Messages</i>

Introduction

This appendix describes the solutions to some of the most common problems for which users call the Technical Support Division. This appendix also includes questions that the Technical Support Division anticipates will be generated by the conversion to Version 6.

Before calling the Technical Support Division, read the `read.me` file on disk 1 carefully. (The `read.me` file is also copied to the directory into which you install the product if you installed the SAS/C Development System on a hard drive.) This file contains errata for the documentation and information about any feature that may have been added after the documentation went to press.

Resolving Undefined Symbols

When I compile and link, I get messages reporting Undefined Symbol: _CXnnn and saying that the proper math library has not been included.

In your program, you have accessed floating-point math routines, but you have not linked with the appropriate math library. Specify the appropriate `math` option for the library with which you want to link, such as `math=standard`. If you are using the `link` option to link your program, the compiler links with the correct math library. If you

call the linker separately from the compiler, you must specify the correct math library in the `slink` command after the `libs` keyword. Refer to *SAS/C Development System Library Reference, Version 6.0* for more information about math libraries.

I am getting BLTN and/or CXERR errors, and I am using math libraries and header files.

You may be including the math header files for one type of floating-point format and linking with libraries for another type of floating-point format. For example, you may be including `m68881.h` and linking with `scmf fp.lib`. Different floating-point formats are incompatible and should not be mixed. Make sure that your included header files match the math library with which you are linking.

If you find no problem with the compatibility between your header files and your math library, contact the Technical Support Division.

When I link my project, I get undefined symbols. This code worked in Release 5.10.

Some of the library symbol names have been changed to comply with the ANSI Standard for symbol names. The ANSI Standard states that any symbols that are not mandated by the Standard must begin with an underscore and a capital letter or two underscores.

When you compile your program, the compiler adds an additional underscore to the beginning of any symbols defined in C code. Therefore, a C symbol with two underscores has three underscores when the linker finally sees the symbol.

If your code refers to a symbol that has changed names from Version 5 to Version 6, the linker may produce an **undefined symbol** message when you link your program. If you receive this message, first check the *SAS/C Development System Library Reference* for different versions of the symbols with different numbers of underscores. Remember, the linker reports one more underscore on the symbol than the *SAS/C Development System Library Reference* shows.

Change your C code to refer to the new name for the symbol as described in the *SAS/C Development System Library Reference*. If this solution is not practical because of the number of references to the symbol, you can solve this problem in one of two other ways:

- Use the **define** option on the `sc` command or a **#define** statement in a header file to define the old name to the new name.
- Use the **define** option on the `slink` command to force all references to one of the names to refer to the other. If you define the item in your code (in other words, you declare it without the **extern** keyword), you should use the **define** option to define the

new name (used by the library) to the old name (used by your code). If you declare the item with the **extern** keyword and, therefore, use the library definition for the item, use the **define** option to define the old name (used by your code) to the new name (used by the library).

One very common technique used in Version 5 was to declare the main routine of your program as **_main** instead of **main** to bypass the overhead required to set up **stdio** and argument parsing. You should change this name to **__main** and add the keyword **__stdcall** to be compatible with Version 6. If you do not change this name, the linker issues a message saying that the symbol **_main** is undefined. (Remember, the linker puts in an extra underscore, so the symbol it is really looking for is **main**).

Fixing Compilation Errors

I have installed my compiler correctly, but I keep getting the message `se not found` or `sc not found` whenever I try to do anything.

Your **s:user-startup** may not be set up correctly. The installation program places three **assign** statements and the **path** statement at the end of your **s:user-startup** file. If you are invoking any programs that create a new Shell (such as PopCLI) in your **s:user-startup** file before the **path sc:c add** statement, **sc:c** may not be in new shells created by the program. To correct the problem, move the three **assign** statements and the **path** statement above any calls to programs that create a new Shell.

If you are running under AmigaDOS Version 1.3, you may have a different problem. The installation program places the necessary **assigns** in a file named **s:user-startup**, which is called from **s:startup-sequence**. You may have to edit **s:startup-sequence** and make some changes. See Chapter 1, "Installing Your SAS/C Development System," for information on modifying your startup file.

After switching to Version 6, my old programs will not compile, or they compile with a lot of warning messages. Why is this happening and how can I fix it?

There are two solutions:

- In Version 6, the compiler assumes that your code will provide prototypes for all functions. (In Version 5, you had to specify the **-cf** option for the compiler to identify missing prototypes.) If your code does not define prototypes for all functions, the compiler may produce several warning 100, 154, and 161 messages. Specify

`ignore=100+154+161` to suppress these warnings, or use the `genprotos` option to generate prototypes for your functions and `#include` the resulting header file.

- Unlike previous versions, the header files and libraries for Version 6 of the SAS/C Compiler are ANSI-compliant. The ANSI specifications affect both ANSI and non-ANSI functions and data names, so some non-ANSI functions and data names have also changed.

Convert your program to use the new ANSI-compliant libraries and headers. For any functions mentioned in error messages, read the description of the function in the *SAS/C Development System Library Reference*. This reference manual describes the parameter list, return value, and necessary header files required by each function. Make sure that you are including the correct header for each of the SAS/C or AmigaDOS functions that you are calling.

When I compile, the options I requested do not act as expected.

When you compile your program, whether you compile from the Shell or the Workbench screen, the compiler looks for an `scoptions` file in your current directory. If it does not find one, it looks for the file `ENV:sc/scoptions`. If it finds either of these files, it uses any additional options specified in the file. You can run the `scopts` utility or edit the `scoptions` file to review the options specified in these files. If you want to prevent the compiler from reading the options specified in any `scoptions` file, specify the `resetoptions` option as the first option in the `sc` command. If your program then runs as expected, you have a problem with the options that you have specified. If you specify the `verbose` option, the compiler tells you the location of the `scoptions` file used.

During compilation, I get the message semi-colon expected in a header file.

Check for extra characters at the beginning of your file in front of your `#include` statements or in one of your header files. You may have accidentally pressed a key while starting the editor.

I keep getting the message Error 25: Modifiable lvalue required, but I am declaring all my variables for a function just like it says in the library reference manual.

Do not include the `const` keyword in your declarations. The *SAS/C Development System Library Reference* contains many variables defined with the `const` keyword, especially pointers to strings. This keyword is required by the ANSI Standard and indicates that these variables will not be modified in the library function to which they are passed. The `const` keyword is present in the parameter list of the prototype

that is found in the associated header file. This prototype, not the declaration you should include in your program, is what is described in the synopsis for each function in the *SAS/C Development System Library Reference*. Use the synopsis as a guideline for your declarations, but do not include the `const` keyword.

Using CodeProbe

I do not get all of the information I expect out of CodeProbe, or CodeProbe does not know about variables and functions it should know about.

You may not be compiling with the correct debug option. If you are compiling with `nodebug` or `debug=line`, try compiling with `debug=sf`. For a complete description of each `debug` option, see Chapter 8, “Compiling and Linking Your Program.”

CodeProbe does not work if I double-click on the Debug icon from the WorkBench screen.

To invoke CodeProbe from the WorkBench screen, click on the **Debug** icon. Then, hold down the Shift key and double-click on the icon for your program’s executable module.

Using Formatted Print Functions

printf (or sprintf, fprintf, and so on) does not print the correct values.

This problem can happen for the following reasons:

- You are asking `printf` to print a variable, but the variable conversion characters are incorrect. For example, you are trying to print a `long integer` using `%d` as the conversion character. A `long` variable requires a `%ld` if the `shortint` compiler option is active. Often, if you have one conversion character wrong, the rest of the line you are printing is also incorrect. Check your entire `printf` (or `fprintf`, etc.) format string carefully.
- You are trying to print a `float` or a `double`, but you have not linked with the proper math library. You must link with a floating point math library to perform any floating point operations. If you are linking with a floating point library, make sure that it is positioned in your `slink` command line before `sc.lib` so that the linker can find the correct version of `printf`. If you specify the `link` option in the `sc` command, the compiler links the libraries in the correct order.

- You are linking with `amiga.lib` before the math library and `sc.lib`. You should specify `amiga.lib` as the last library in the `slink` command.

Using `getchar`

`getchar` does not work as expected.

Input and output on the Amiga system are buffered. On most other systems, `getchar` immediately gets a character from `stdin` (usually the keyboard). However, the `CON:` device on the Amiga system buffers its input, so your program does not actually read any characters until you do one of the following:

- fill up the console input buffer
- press Return
- enter the Amiga End-of-File character, Ctrl-\..

The SAS/C Development System libraries include two functions that you can use to deal with this problem:

`getch` gets a character from the console in **RAW** mode. The character is returned as soon as the user types it.

`rawcon` turns **RAW** mode on and off. `rawcon(1)` sets the console into **RAW** mode. `rawcon(0)` restores the console to non-**RAW** mode.

When the console is in **RAW** mode, any characters typed by the user are passed immediately to the application.

`getch` returns a single character from `stdin` just like `getchar`, but `getch` gets the character in **RAW** mode. If your console is not in **RAW** mode, using `getch` is equivalent to the following sequence:

```
rawcon(1);
c = getchar();
rawcon(0);
```

If your console is in **RAW** mode, `getch` is equivalent to `getchar`.

For more information about `getchar`, refer to *SAS/C Development System Library Reference*.

Getting Incorrect Results from Function Calls

My program compiles and links without errors, but I am getting the wrong results from some of my SAS/C function calls.

You can get incorrect results from a function if you do not include the correct header files for the function you are calling. The header files contain prototypes for each of the SAS/C functions, as well as definitions for many common data structures required by these functions. By including the correct header file, you are providing the compiler with a prototype for the SAS/C functions you call.

The ANSI Standard requires that the compiler assume that functions that are called without the previous inclusion of a prototype return an `int`. For example, this means that if you call `atof`, which returns a `double`, without including `math.h`, the compiler assumes that `atof` returns an `int` and allocates only 4 bytes for the return value. When `atof` is called and the correct 8-byte `double` value is returned in the 4-byte space allocated for it, the value appears to be incorrect.

You may be including the incorrect header file for one of these reasons:

- In Version 6, the prototype is contained in a different header file than in Version 5. For example, in Release 5.10, the function `atof` was included in `math.h`. In Version 6, the `atof` prototype was moved to `stdlib.h` as required by the ANSI Standard. Check the *SAS/C Development System Library Reference* for the correct header file for the functions that you are using.
- You may be including the right header file from the wrong directory. For example, the `include:dos` directory contains many header files with the same names as those in the root level `include:` directory. However, these header files are AmigaDOS header files and not ANSI header files. The AmigaDOS header files do not contain the necessary prototypes and data declarations. For example, you may be including `dos/stdio.h` instead of `stdio.h`. Do not substitute header files from the `include:dos` directory for standard headers.
- You may be including header files from the `proto` directory instead of those in the root level `include:` directory. Although the files in the `proto` directory do contain prototypes for many of the SAS/C functions, these files are intended to be included in addition to the standard header files, not in place of them. Do not use header files in `include:` subdirectories as substitutes for the root level header files.

Crashing the Machine

My program compiled and linked without any errors, but when I run it, my machine crashes. What am I doing wrong?

The following list contains some of the more common errors that crash programs. Most of these errors are caused when your program accesses memory that it is not supposed to access.

- You have declared a pointer to an array or structure for which you have not allocated memory. This mistake is especially easy to make when calling functions that fill the array or structure with information, such as the `fstat`, `lstat`, and `stat` functions.

To correct this problem, call `malloc` or `calloc` to allocate the appropriate amount of memory for the array or structure.

- You have assigned an inappropriate value to a pointer, so the pointer no longer points to a valid memory space. This type of error can be quite difficult to track down. You may want to run the CodeProbe, the Enforcer, or the Mungwall program if you think these types of errors may occur. Enforcer and Mungwall are programs that detect memory errors. They are available from Commodore Applications and Technical Support (CATS) and are shipped with Version 6.
- You have specified the `nostackcheck` option and are overwriting your stack. Do not specify `nostackcheck` until your program is completely debugged. The `stackcheck` option is the default unless you are creating a shared library using the `libcode` option.

Using The `asctime` Function

I am using the `asctime` function to convert the time to Greenwich Mean Time (GMT) correctly, but the result is exactly an hour (or two or three...) off.

Your machine is probably in the default time zone, Central Standard Time (CST), and has not been reset to your actual time zone. To correct the problem, reset your machine's time zone environment variable to your actual time zone with the following AmigaDOS command:

```
setenv TZ=your-time-zone
```

For *your-time-zone*, enter your zone's standard three letter abbreviation followed by the number of hours difference between your time zone and Greenwich Mean Time. For example, for Eastern Standard Time, the command would be as follows:

```
setenv TZ=EST5
```

You can also initialize the `_TZ` variable as described in the *SAS/C Development System Library Reference*. Initialize the `_TZ` variable with the same time zone data as you would enter with the AmigaDOS command described above (for example, EST5).

Note: The AmigaDOS environment variable does not have a leading underscore that the SAS/C data name has.

Managing the Standard I/O Window

An annoying window opens when I run my program from the Workbench screen. How do I get rid of this window?

The SAS/C startup code opens the window to allow programs that are run from the Workbench screen to access `stdin`, `stdout`, and `stderr`. You can eliminate this window or change its attributes. For information on managing this window, see Chapter 9, "Running Your Program from the Workbench Screen."

Linking Resident Programs or Creating Resident Libraries

I get the message `absolute reference to name from the linker`.

You are linking a resident program with the `cres.o` startup module, or you are creating a resident library, but the linker detected a reference to an absolute symbol. Make sure that the symbol being referenced is not being modified. If the item is not being modified, then you can ignore the warning message.

Writing Replacement Functions for SAS/C Library Functions

I wrote a replacement function for one of the SAS/C library routines, but the library routines don't seem to be using my replacement function.

If you write a function that has the same name as a library function, you need to add the `__regargs` keyword to your function or compile with the `parms=both` or `parms=register` option. For example, you may want to replace the SAS/C library function `malloc` with your own version of `malloc`. If you compile with the `parms=stack` option or define your version of `malloc` with the `__stdargs` keyword, then two versions of `malloc` are linked into your executable. If you use other SAS/C library functions that call `malloc`, these functions use the version of `malloc` in the SAS/C libraries. However, your functions that call `malloc` use your version of `malloc`. To make sure that all calls to `malloc` are using your version of `malloc`, define your version with `__regargs` or compile with the `parms=both` or `parms=register` option.

Eliminating Informational Messages

How can I turn off all of the informational messages the compiler and linker produce every time I compile and link?

You can specify the following options on the `sc` command line:

```
sc nover link filename.c
```

Appendix 2

Error and Warning Messages

203	<i>Introduction</i>
203	<i>Using Error and Warning Messages</i>
204	<i>Explanations of Unnumbered Compiler Messages</i>
206	<i>Explanations of Numbered Compiler Messages</i>
258	<i>Explanations of Linker Messages</i>
265	<i>Enabling Suppressed Messages</i>

Introduction

This appendix explains each of the error and warning messages that may be produced by the compiler and linker.

Using Error and Warning Messages

You should treat warnings as seriously as errors. If you do not know why a given warning is being produced, find out why. If you can safely ignore the warning in the future, use the `ignore` option on the compiler or the `#pragma msg` statement to suppress the warning.

In general, the compiler attempts to prevent long cascades of error and warning messages that result from a single mistake, but this action is not always possible. As a result, you may see several messages generated by the same error, especially if the error is in a control statement such as an `if`, `for`, `do`, or `switch` statement. If you receive some messages that seem to be incorrect or confusing, fix as many errors or warnings as possible, and then recompile your program. The confusing messages may disappear.

Sometimes the compiler may generate an error message for code that you believe is correct. In many cases, such errors are the result of incorrectly using the preprocessor. A typographical error in a preprocessor macro or accidental collision of a `#defined` macro name with another name in your program can cause very confusing problems. If you think you are having problems with preprocessor macros, compile your program with the `ponly` option to generate preprocessed output, and check that to see exactly what the compiler is receiving. Alternatively, compile with the `list` option to generate a listing file. The

listing file allows you to see exactly what symbols are being defined by the header files included by your program.

Some error or warning messages are produced at the first non-preprocessor statement after the actual condition that caused the error or warning. For example, if your program is missing a semicolon, you do not get a message at the end of the line that is missing the semicolon; you get a message at the beginning of the next line of code. If you cannot find an error exactly where the message occurred, look at several previous lines in your source code. Remember, the compiler treats **#include** files as if they are part of your source file. For example, a missing close curly brace at the end of your last **#include** file may result in an error message at the first line of your C source file. You can compile with the **pponly** option to help diagnose this type of error.

By default, many warning messages are suppressed but can be enabled with the **strict**, **ansi**, or **warn** options. The descriptions of each message indicate whether the message is suppressed and, if so, which options enable the message. You can enable any suppressed warning with the **warn** or **error** options. You can disable any warning with the **ignore** option. You cannot use the **ignore** option to disable error messages. See Chapter 8, “Compiling and Linking Your Program,” for more information about the **strict**, **ansi**, **error**, **warn**, and **ignore** options. See also “Enabling Suppressed Messages” later in this appendix for a complete list of the suppressed messages and the options you can use to enable them.

Some messages listed as warnings can also be produced as errors in some cases. If a message is produced as an error instead of a warning, it cannot be ignored (just as errors cannot be ignored).

You can display help information on a specific message from the message browser utility (**scmsg**) by clicking on the message and pressing the Help key or by invoking the AmigaGuide utility on the file **sc:help/scmsg.guide**.

Explanations of Unnumbered Compiler Messages

Freeing Resources

If you compile your program and your machine runs low on memory, the compiler displays this message and frees memory to enable it to continue the compilation. You can force the compiler to free memory at any time by pressing Control-F in the window to which the compiler is sending output.

Floating point overflow optimizing constants

The global optimizer was attempting to perform compile-time constant calculations, but the calculations caused a floating-point error. Your code is causing floating-point numbers to overflow.

Can't open type file "name" for mode

The compiler could not open the specified file. The **mode** is either **read** or **write**.

Combined output filename too long

The filename produced by the compiler, with the path, overflowed the compiler's internal buffer (255 bytes).

Can't open sc:libs/lib-name.library

The specified shared library could not be found. Make sure the library is available in **sc:libs**.

CXWRN: text

An internal error occurred. With **CXWRN** errors, the compiler attempts to continue the compilation, but may not be able to do so. Please contact the Technical Support Division.

CXERR: num

An internal error prevented the compiler from continuing. Please contact the Technical Support Division.

Can't delete old GST: object is in use**Continuing with no GST file**

You specified the **makegst** option, but an existing copy of the same GST was in use by another program. Check for other compilations that are using the GST. You may also be browsing the GST with the **hypergst** utility.

Can't open GST file: gst-filename

The specified GST file could not be loaded. Either the file is an invalid GST file, or the file does not exist.

Invalid symbol definition: symbol-name

You attempted to define the specified symbol on the command line with the **define** compiler option, but the symbol did not adhere to the normal rules for C preprocessor symbol syntax.

Seek error on object file

The compiler attempted to perform a **seek** operation on the output object file but encountered an I/O error.

I/O error code on file "name"

The compiler received the specified I/O error code from the operating system while attempting to read the named file. Refer to *The AmigaDOS Manual, 3rd Edition (Commodore-Amiga, Inc. 1991)* or see the header file **dos/dos.h** for details on the numeric error codes.

Explanations of Numbered Compiler Messages

Warning 1: `invalid preprocessor command`

This warning is generated by invalid use of preprocessor commands. For example, you could specify an unrecognized command, fail to include a space between command elements, or use an illegal preprocessor symbol. The command is ignored and compilation continues.

Error 2: `unexpected end of file`

This error is generated when the compiler expects more data, but it encounters the end of an input file. This error may occur in a `#include` file or in the original source file. A missing `#endif` or unbalanced curly brace or parentheses in the source file or in one of the previously included files may produce this message. In many cases, correcting a previous error eliminates this error.

Error 3: `file not found "filename"`

The filename specified in a `#include` command was not found or could not be opened.

Error 4: `invalid lexical token`

A character was found in the file that is not a standard character in the C character set or is in an inappropriate place. For example, entering a pound sign (`#`) in the middle of non-preprocessor C code or entering nonprintable control characters anywhere except in a comment produces this error.

Error 5: `invalid macro usage`

Your code invoked a macro incorrectly. Check for unbalanced parentheses and other syntax errors. The problem may be in a macro used by the macro that you invoked in your program.

Error 6: `line buffer overflow`

A line of preprocessed input was longer than the line buffer size. The size of the line buffer is controlled by the `ppbuf` and `memsize` options. If you do not specify `ppbuf` or `memsize`, the size of the line buffer is 8192 bytes.

Error 8: invalid conversion

You attempted to cast a type to an incompatible type. This error usually occurs when you attempt to convert something into an array, a structure, or a function. Check for missing indirection (*) and/or address (&) operators. For example, the following program tries to assign a whole structure to a pointer.

```
void main(void)
{
    struct FOO
    {
        int a, b;
    } f;
    struct FOO *p;
    p = f; /* Error 8 */
    p = &f; /* Correct */
}
```

Error 9: undefined identifier "name"

The specified identifier has not been declared. You may not have included the proper header files to declare an **extern**, or you may have misspelled the name of a variable. This message is produced only once for each undeclared identifier. Subsequent uses of the identifier do not produce a message. Subsequent declarations of the identifier may produce messages about redeclaring the variable. Fix the error that is causing the first error 9 message and recompile your program before trying to fix additional messages involving the same variable.

Error 10: invalid subscript expression

An error was detected in an expression used inside square brackets ([]). This error may occur if:

- ☐ the expression is missing
- ☐ the expression is a preprocessor macro that evaluates to nothing
- ☐ the result of the expression is **void**
- ☐ the result of the expression is a pointer, a structure, or a union.

Error 11: string not terminated

The closing double quote (") was not provided when defining a string.

Error 12: invalid structure reference

The operand preceding the structure member (.) or structure pointer (->) operator is not a structure or a pointer to a structure, respectively. Make sure you are not trying to reference a structure member with the structure pointer operator or a structure pointer with the structure member operator. In many cases, correcting a previous error eliminates this message.

Error 13: member name missing

The name of the desired structure or union member did not follow the structure member operator (.) or the structure pointer operator (->). Check for preprocessor macros that may be defined to the same name as the member.

Error 14: undefined member "name"

The indicated identifier is not a member of the structure or union to which the structure member operator (.) or the structure pointer operator (->) referred.

Error 15: invalid function call

An identifier or constant is used where a function or function pointer identifier is required. This message can occur if you attempt to use a variable not declared as a function or function pointer but give the variable a parameter list. The compiler sees the parenthesized expression and thinks you are calling a function. Check for typographical errors such as leaving out an operator in an expression, which produces a variable and a parenthesized expression. Such typographical errors may also occur in preprocessor macros. For example:

```
/* The incorrect expression below generates an */
/* "invalid function call" error.                */

int i, j;
void function(void)
{
    i = i + (j*2); /* Intended expression */
    i = i   (j*2); /* INCORRECT - deleted "+" operator */
}
```

Error 16: invalid function argument

A function argument expression following the left parenthesis of a function call is invalid. You may see this message if you omit:

- ☐ an argument expression
- ☐ a right parenthesis from a function call
- ☐ a comma separator between two function arguments.

For example:

```
func(.);
func();
func(,1);
```

Error 17: too many operands

During expression evaluation, the end of an expression was encountered, but more than one operand was still awaiting evaluation. This message may occur if an expression contained an incorrectly specified operation.

Warning 18: non-ANSI use of operator in preprocessor condition

This message is suppressed by default, but you can enable it with the **strict**, **ansi**, or **warn=18** options. The ANSI Standard states that the **sizeof** and comma **(,)** operators should not be used in preprocessor conditions. The SAS/C Compiler supports their use in preprocessor conditions, but programs that require strict adherence to the ANSI Standard should not use them.

Error 19: unbalanced parentheses

The number of opening parentheses in an expression did not equal the number of closing parentheses. If the expression appears correct in the C source file, check any preprocessor macros to make sure they generate balanced parentheses.

Error 20: invalid constant expression

An expression that did not evaluate to a constant was encountered in a context that required a constant result. The compiler must be able to evaluate any constant expression (for example, expressions used to initialize **static** or **external** data) when the program is compiled. The expression in question did not meet this criterion. You may have used an illegal operator for a constant expression (such as **++**, **+=**, function calls, and so on), or you may have used a variable whose value is available only at run time.

Error 21: illegal use of struct, union, or array type

An identifier declared as a structure, union, or array was encountered in an expression where such types are not permitted. For example, you cannot use the ++ postincrement operator with a structure:

```
struct F00
{
    int a, b, c;
} f;

f++;      /* Error 21 */
```

Warning 22: structure used as function argument

This message is suppressed by default, but you can enable it with the **warn=22** option. A structure or union was passed as an argument to a function. Although passing structures and unions is legal according to the ANSI Standard, some compilers do not allow you to pass structures or unions. Other compilers pass a pointer to the structure instead of the entire structure. Turn on this warning if your code comes from or must be ported to such a compiler.

Error 23: invalid use of conditional operator

The conditional expression operator (?:) was used incorrectly. You may have included the question mark (?) but left out the colon (:). Also, check for an invalid expression after the operator.

Error 24: pointer operand required

An expression required a pointer at a specific place, but a non-pointer operand was provided. The compiler may generate this message if an expression after the indirection operator (*) was not a pointer or array expression or if the expression before the array indexing operators ([]) was not a pointer or array expression.

Error 25: modifiable lvalue required

You have attempted to assign a value to an expression that cannot be modified. An *lvalue* is any expression that can appear on the left side of an assignment operation. For example, the ANSI Standard states that the result of a cast is not an lvalue; therefore, the following statement is invalid:

```
long x;
(short)x = 2; /* Error 25 */
```

The following examples also generate this error message:

```
#define ADDONE(x) (x)++
ADDONE(12);      /* Error 25: Cannot increment a constant */
ADDONE(&g);      /* Error 25: Cannot assign to an address */

if(func(10)=j-2); /* Error 25: "=" was intended, not "=" */

&x = &y;         /* Error 25: Cannot assign to an address */
```

You may have defined a variable with the `const` keyword and then tried to modify the value of that variable. The *SAS/C Development System Library Reference, Version 6.0* contains many variables defined with the `const` keyword, especially pointers to strings. This keyword is required by the ANSI Standard and indicates that these variables will not be modified in the library function to which they are passed. The `const` keyword is present in the parameter list of the prototype that is found in the associated header file. This prototype, not the declaration you should include in your program, is what is described in the synopsis for each function in the *SAS/C Development System Library Reference*. Use the synopsis as a guideline for your declarations, but do not include the `const` keyword.

Error 26: arithmetic operand required

An expression required an arithmetic operand but the provided operand was not arithmetic. An operand is arithmetic if it declared as `char`, `short`, `int`, `long`, `float`, or `double` or the signed and unsigned variants of these types.

Pointers, structures, unions, and functions are not arithmetic operands.

Error 27: arithmetic or pointer operand required

An expression required an arithmetic or pointer operand, but a structure or union was provided. An operand is arithmetic if it declared as `char`, `short`, `int`, `long`, `float`, or `double` or the signed and unsigned variants of these types.

A pointer operand can be a pointer to any other data type, or it can be the address of a variable or function.

Error 28: missing operand

During expression evaluation, the end of an expression was encountered but not enough operands were available for evaluation. The compiler may generate this message if you specified a binary operator (such as the addition, subtraction, multiplication, or division operator) with only one operand. Also, check for invalid preprocessor macro expansions. For example:

```
int i;
int ary[10];

i = i + ;      /* Error 28 */
i = ary[i*]4; /* Error 28 (Among others) */
```

Error 29: operation cannot be performed on a pointer

An operation was specified that was invalid for pointer operands, such as one of the arithmetic operations other than addition or subtraction.

Warning 30: pointers do not point to same type of object

In an assignment statement defining a value for a pointer variable, the expression on the right side of the assignment (=) operator did not evaluate to `NULL` or to a pointer of the same type as the pointer variable on the left side of the assignment operator. The warning is also produced when a pointer of any type is assigned to an arithmetic object.

Error 31: integral operand required

An expression required a given operand to be an integral type, but the actual operand was not an integral type. An operand is integral if it is declared as `char`, `short`, `int`, or `long` or the signed and unsigned variants of these types.

For example, the following code generates error 31:

```
double d;
int ary[10];

ary[d] = 10; /* Error 31 */
```

Error 32: cannot convert to required type

The compiler was unable to convert a data item from its base type to the type required by the operation. The compiler may generate this message if you attempt to cast any data type to a structure, instead of casting the type to a pointer to a structure, or if you attempt to cast a structure to any other type. This message can also be produced for implied conversions, such as passing a structure as a parameter to a function expecting some other type. For example, the following code generates error 32:

```
struct F00 f;
int j;

j = (int)f; /* Error 32 */
```

Warning 33: non-portable operation on structure or union

Your code has attempted to use an operator on a structure or union that is illegal for that type. For example, you may have used the equality == operator on two structures. The ANSI Standard does not permit the use of relational operators on structures or unions.

The SAS/C Compiler generates the equivalent of a `memcmp` call for this construct, but it may not perform as expected. Because structures may contain padding bytes, two structures of the same type with all identical members may compare false. If you intend to use direct structure comparison, make sure you declare the structure `static` or `extern`, or initialize the structure to zeroes using a call to `memset`.

For example, the following code generates warning 33:

```
#include <proto/dos.h>
struct FileInfoBlock fib1, fib2;

if(fib1 == fib2) /* Warning 33 */
```

Error 34: invalid initializer expression

The expression used to initialize an object was invalid. The compiler may generate this message if you fail to separate elements in an initializer list with commas or if you attempt to initialize an array to a single object, as shown in the following examples:

```
int a[3] = 0; /* Error 34 */
int b[3] = { 1 2 3 }; /* Error 34 */
```


Error 35: closing brace expected

The compiler expected a closing brace (}) to terminate the definition of a function, structure, or nested block scope, but the brace is missing. The compiler may generate this message if:

- ☐ too many elements occur in an initializer expression list
- ☐ a structure member was improperly declared
- ☐ the end of the source file is reached before a definition is complete
- ☐ a previous error occurred in a control statement.

Warning 36: control cannot reach this statement

A statement with no label followed a **goto**, **return**, **break**, or **continue** statement. The statement is therefore unreachable. This warning can sometimes be produced incorrectly if the compiler reported a previous error while in a control flow statement. Fix all previous errors and recompile your program before trying to fix the error that is generating this message.

Error 37: duplicate statement label "*name*"
See line *number* in file "*filename*"

The specified statement label has already been defined in the current function. You cannot define the same label more than once in the same function.

Error 38: unbalanced braces

In a set of compound statements, the number of opening left braces ({) did not equal the number of closing right braces (}). This warning may be produced incorrectly if the compiler reported a previous error in a control flow statement. Fix any previous errors and recompile your program before trying to fix the error that is generating this message.

Error 39: invalid use of keyword "*keyword*"

One of the C language reserved words appeared in an invalid context (for example, as a variable name).

Error 40: break not inside loop or switch

A **break** statement was detected that was not within the scope of a **while**, **do**, **for**, or **switch** statement. This error may be produced incorrectly because of errors in previous statements.

Error 41: case not inside switch

A **case** prefix was encountered outside the scope of a **switch** statement. This error may be produced incorrectly because of errors in previous statements.

Warning 42: case expression not integral

The expression defining a **case** value did not evaluate to an integral constant. This message is generated as an error message if the expression could not be converted into an integral constant and as a warning if the expression could be converted into an integral constant. For example, if you use a variable as a **case** value, the compiler generates an error message. If you use a floating-point constant as a **case** value, the compiler converts the constant to an **integer** (thereby truncating its value) and generates a warning message. For example, in the following code, the **case** value 1.6 is truncated to 1, and the value -1.6 is truncated to -1 .

```
switch(i)
{
    case 1.6: /* Warning 42 */
        break;

    case -1.6: /* Warning 42 */
        break;
}
```

Error 43: duplicate of case value See line number in file "filename"

You have used the same **case** value more than once within the same **switch** statement. Check for possible preprocessor macro definitions that expand to the same value. For example, both of the following **case** statements evaluate to zero:

```
#define FOO 0
#define BAR 2

switch(i)
{
    case FOO:
        break;
    case (FOO*BAR): /* Error 43 */
        break;
}
```

Error 44: continue not inside loop

A **continue** statement was detected that was not within the scope of a **while**, **do**, or **for** loop. This error may be produced incorrectly because of errors in previous statements.

Error 45: default not inside switch

A **default** label was encountered outside the scope of a **switch** statement. This message may be produced incorrectly because of errors in previous statements.

Error 46: duplicate default See line number in file "filename"

A **default** label was encountered within the scope of a **switch** statement in which a **default** label had already been encountered.

Error 47: while missing from do statement

A **while** clause did not follow the body of a **do** statement. This message may be produced incorrectly because of errors in previous statements.

Error 48: invalid while expression

The expression defining the looping condition in a **while** or **do** loop was **void** or was missing. If you intend for a loop to be infinite, you must supply a constant (such as 1) for the **while** condition. The error may be produced because of a preprocessor macro that expands to invalid code. In many cases, fixing a previous error eliminates this message.

Error 49: else not associated with if

An **else** keyword was detected that was not in the scope of a preceding **if** statement. This message may be caused by an error in a preceding statement, especially if the previous error occurred while processing the **if** statement with which the **else** statement is associated.

Error 50: label missing from goto

The compiler expected a statement label to follow the **goto** keyword but the label was missing. This message may be produced incorrectly because of errors in previous statements.

Warning 51: C++ comment detected

This message is suppressed by default, but you can enable it with the **strict**, **ansi**, or **warn=51** options. In C++, comments begin with two slashes (//) and terminate at the end of the line on which they begin. The SAS/C Compiler accepts comments entered in this way for convenience and for compatibility with other implementations, but they are not part of the ANSI Standard for the C language.

Error 52: invalid if expression

The expression following the **if** keyword on an **if** statement was **void**, invalid, or missing. This error may be caused by a preprocessor macro expanding to inappropriate values or may be the result of an expression with an invalid return type (such as a structure type). In many cases, fixing a previous error eliminates this message.

Error 53: invalid return expression

The expression following the **return** keyword was **void**, invalid, or missing. The compiler may generate this message if a preprocessor macro expands to inappropriate values. In many cases, fixing a previous error eliminates this message.

Warning 54: switch expression not integral

The expression defining the test value for a **switch** statement did not define an integral value as required by the ANSI Standard. The value supplied is converted to **int** before any attempt is made to use it. If the **switch** value is a floating-point value, this conversion may truncate the value. This warning can also be generated as an error if the value could not be converted to **int**.

Warning 55: no case values for switch statement

The statement defining the body of a **switch** statement did not define any **case** statements. This warning may be produced incorrectly because of previous errors.

Error 56: colon expected

The compiler expected but did not find a colon (:). This message may be generated if a **case** expression was improperly specified or if the colon was omitted following a label to a statement. Because the compiler scans through newlines, blanks, and comments looking for the colon, this message is usually produced at the beginning of the line following the actual error.

Error 57: semi-colon expected

The compiler expected but did not find a semi-colon (;). This error can be generated if you use too many right parentheses or right curly braces (}). Because the compiler scans through spaces and tabs looking for the semicolon, this message is usually produced at the beginning of the line following the actual error.

Error 58: missing parenthesis

A required parenthesis is missing. This error is often caused by previous errors.

Error 59: invalid storage class

Possible storage classes are denoted by the keywords `__near`, `__far`, `__chip`, `register`, `auto`, `extern`, and `static`. Some of these keywords are invalid for certain types of data. For example, you cannot declare an external variable with the `register` keyword, and you cannot declare an automatic (local) variable with the `__chip`, `__near`, or `__far` keywords. Your code attempted to use a storage class keyword incorrectly. This error often occurs because of previous errors.

Error 60: incompatible struct, union or array types

Incompatible structure, union, or array types were used in an expression.

```
struct A {int x;}    a;
struct B {double d;} b;
a = b; /* Error 60 */
```

Error 61: undefined struct/union tag "tag-name"

Your code has used a structure or union tag that has not been declared. Check for misspelled structure names. You may want to compile your program with the `pponly` option or the `list` option and look at the output produced.

Warning 62: constant *number* out of range for type "*type*"
 Valid range is *low* to *high*

The constant value indicated is not in the range of possible values for the type to which the value is being assigned. This message may occur if you are assigning a hexadecimal constant with the uppermost bit set to a signed variable. By definition, a hexadecimal constant is a positive value; therefore, the assignment to the variable reinterprets the constant as a negative number. For example, the following lines generate warning 62:

```
signed char c = 0x80;    /* Warning 62 */
short s = 100000;       /* Warning 62 */
unsigned short uc = -1; /* Warning 62 */
```

Unless you use the `unchar` compiler option, variables of type `char` are signed and produce this warning if you assign any constant to them with the high bit set (that is, any value between 128 and 255.) You can suppress this warning for a specific case by casting the constant to the appropriate type.

Out-of-range constants can cause problems in your code that are hard to debug. For example, the following code does not behave as intended:

```
int i = 0xff;
signed char c = 0xff; /* Warning 62 */

if(i == c)
{
    /* Not executed */
}
```

The constant initializer `0xff` is the decimal number 255. When assigned to the integer variable `i`, this value is preserved, and `i` gets the value 255. When assigned to the signed character variable `c`, the value of `c` becomes `-1` because a signed character cannot represent numbers higher than 127, and the constant `0xff` overflows. Therefore, the comparison in the `if` statement results in the value false.

Warning 63: item "*name*" already declared
See line *number* file "*filename*"

The named item was previously declared at the cited location. This warning is produced when two different members of the same structure, union, or `enum` are given the same name.

Error 64: structure contains no members

A structure declaration did not contain any members. This error can be produced by errors encountered during the structure's declaration. Fixing a previous error may eliminate this message.

Error 65: invalid function definition

Your code tried to define a function body inside another function body, inside a structure declaration, or inside a list of static initializers. This message may be produced incorrectly because of errors in previous statements.

Warning 66: invalid array limit expression

The expression defining the size of the subscript in an array declaration did not evaluate to a positive integral constant. For example:

```
int x[10.7];
```

Error 67: illegal object

Your code attempted to define an illegal item. For example, you may be declaring an array of functions (instead of an array of function pointers) or a function that returns an array (instead of a function that returns a pointer). This message can also be produced if you attempt to define a variable with the same name as a keyword. For example, the following code generates error 67:

```
int break; /* Error 67 */
long while; /* Error 67 */
```

Error 68: illegal object for structure

A structure (or union) included a function as a member. You cannot include a function as a member of a structure or union, although you can include a function pointer.

Error 69: struct name includes instance of self

The named structure or union contains an instance of itself. Although it is legal for a structure or union to include a pointer to its own type, the structure or union cannot contain an instance of itself. If the structure

or union does not have a name, the *name* field is not printed in the message. For example, the following code generates error 69:

```
struct F00
{
    int a, b, c;
    struct F00 x; /* Error 69 */
};
```

Warning 70: unrecognized escape sequence

By default, this message is suppressed, but you can enable it with the **strict**, **ansi**, or **warn=70** options. Escape sequences in string and character constants begin with a backslash (\) and contain one or more characters after the backslash. The ANSI Standard defines some escape sequences and reserves others for future expansion and for implementation-defined extensions. The SAS/C Compiler ignores the backslash on any such undefined escape sequences. Other compilers may take different action. For example, the following line prints the character **q** followed by a newline (\n) to **stdout**:

```
printf("\q\n"); /* Warning 70 */
```

Other ANSI-conforming compilers may substitute any other character for the **\q** escape sequence.

Error 71: formal declaration error "name"

A variable was declared before the left curly brace of in an old-style function definition, but the variable did not appear in the list of identifiers in parentheses following the function name. For example, the declaration of **j** in the following code generates error 71:

```
int func(i)
int i;
int j; /* Error 71 */
{
}
```

You may have misspelled one of your formal parameters or forgotten to add the parameter to the parameter list.

Error 72: conflict with previous declaration
See line number file "*filename*"

A variable or function was declared that conflicts with a previous declaration for the variable or function in the same scope. The message indicates the filename and line number of the original declaration. If no prototype exists for a function, the first use of that function implicitly declares the function as returning an `int`. If the actual definition of the function follows the first use of the function and declares a different return type, the compiler produces this message. This message may be produced incorrectly if you forget to declare a variable in an earlier location and, therefore, received a previous error message about an undefined identifier with the same name. Fixing the previous error may eliminate this message.

Warning 73: declaration expected

The compiler expected to find the declaration of a data object or function but did not. This message can also occur if you enter too many or too few curly braces. This message may be produced incorrectly because of errors in previous statements. Fixing the previous errors may eliminate this message.

Warning 74: initializer data truncated

A static initializer expression contained more elements than the data item being initialized, as in the following example:

```
char b[3] = "abcd";    /* Warning 74 */
```

String constants always have an implied `NULL` byte at the end. This byte is not counted when producing this warning. If you have a character array with three elements, you may initialize it as follows:

```
char b[3] = "abc";
```

The three elements receive the values `a`, `b`, and `c`. If there is room, the `NULL` terminator byte is also copied:

```
char c[4] = "abc";
```

The elements of the above array are assigned the values `a`, `b`, `c`, and `\0`.

Error 75: invalid sizeof expression

The expression passed to the `sizeof` operator was invalid. This message may be produced if an attempt is made to take the size of a function, bitfield, or incomplete type, or if the result of the expression used is of type `void`. For example, in the following code, `a` is an incomplete type because its size is not specified when it is declared:

```
extern int a[];
int i;

i = sizeof(a); /* Error 75 */
```

Error 76: left brace expected

The compiler expected but did not find an opening left brace. For example, you may have omitted the opening brace on a list of initializer expressions for an aggregate.

Error 77: identifier expected

The compiler expected to find the name of an identifier to be declared. The compiler may generate this message if the prefixes to an identifier in a declaration (parentheses and asterisks) are incorrectly specified or if a sequence of declarations is listed incorrectly, as in the following example:

```
int ; /* Error 77 */
```

Error 78: undefined statement label "name"

A `goto` statement referred to the named label, but the label does not exist in the function that referred to it. Check the spelling of your label and the `goto` reference. Make sure the label is in the same function as the `goto` statement.

Warning 79: duplicate of enumeration value
See line line file "file"

When declaring an enumeration type, more than one enumeration constant was assigned the same value, as in the following example:

```
enum COLORS
{RED=1, BLUE=2, GREEN=1}; /* RED and GREEN are identical */
```

Any enumeration constants that are not assigned an explicit value are assigned values one higher than the previous constant in that enumeration. If the constant is the first constant in that enumeration, it is assigned the value zero. Therefore, the following `enum` generates warning 79:

```
enum COLORS {RED, BLUE, GREEN=1}; /* Warning 79 */
```

RED is the first constant, so it is assigned a value of 0. BLUE is assigned a value of 1. GREEN is explicitly assigned a value of 1, which conflicts with BLUE.

Warning 80: invalid bit field or misplaced ':'

This warning is commonly produced if you type a colon (:) when a semicolon (;) was expected. This warning can also be produced if you are actually declaring a bitfield and give an improper expression for the number of bits in the bitfield.

Error 81: preprocessor symbol loop; macro expansion too long or circular

When using the `oldpp` compiler option, a preprocessor symbol expanded to a value that contains a circular reference back to the symbol itself, thereby creating an infinite loop in the preprocessor. Check the definition of the macro being expanded on the line in question. This message cannot occur if the `oldpp` compiler option is not used because the ANSI Standard prohibits the expansion of preprocessor macros that occur as a result of expanding a previous instance of the same macro.

Error 82: maximum object/storage size exceeded

Your code attempted to declare a data item that exceeded the maximum size of objects in its storage class, or the last object declared caused the total size of declared objects in that storage class to exceed the maximum, as in the following example:

```
char __near a[30000];
char __near b[30000];
char __near c[30000];
```

Because there is a limit of 32767 bytes on the amount of near data allowed, the above request for 90000 bytes of near data will not work. If you are compiling with `data=near` (the default), the `__near` keyword is implied on all data items. Fix this warning by declaring some large data items with the `__far` keyword or by compiling with the `data=far` option.

Warning 83: reference beyond object size

Your code used an address beyond the size of the object used as the base for the address calculation. This warning usually occurs when you refer to an element beyond the end of an array, as follows:

```
char c[10];
void myfunc(void)
{
    c[11] = 0; /* Warning 83 */
}
```

This message can be produced only when the compiler can determine the value of the subscript at compile time, that is, the subscript is a constant.

Warning 84: redefinition of pragma or preprocessor symbol "name"

See line *number* file "*filename*"

Your code is redefining the preprocessor symbol or **#pragma** originally defined at the indicated file and line number.

**Warning 85: return value mismatch for function "name"
Expecting "type1", found "type2"**

The expression specifying the value to be returned from a function was not the same type as the function itself. If possible, the value specified is converted to the appropriate type. You can suppress this warning by casting the return expression to the appropriate type.

Some pre-ANSI C code produces many of these warnings when functions declared as **int** (either explicitly or implicitly) do not return a value. Before the ANSI committee approved the **void** keyword, declaring functions as returning an **int** was the correct way to handle functions that returned nothing. You can compile with the **nowarnvoidreturn** option to suppress these warnings when a function that is declared as returning an **int** actually returns nothing.

For example, the following two functions generate warning 85:

```
function1(x)
int x;
{
} /* Warning 85 */

int function2(x)
int x;
```

```

{
    char *p = NULL;
    return(p); /* Warning 85 */
}

```

You can suppress the warning 85 for `function1` by compiling with the `nowvret` option. The `nowvret` option does not affect `function2`, which is attempting to return a pointer from a function declared as returning `int`.

Warning 86: formal parameters conflict with prototype
See line *number* file "*filename*"

The types of the parameters to the function do not match the types given in the prototype for the function. Check the prototype at the file and line indicated against your definition.

Warning 87: argument count incorrect, expecting *number* arguments
See line *number* file "*filename*"

Your code invoked a function with an incorrect number of arguments, according to that function's prototype. Check the prototype at the indicated file and line number against your usage of the function and the actual function definition.

Warning 88: argument type incorrect
Expecting "*type1*", found "*type2*"

Your code invoked a function with an argument whose type conflicts with the corresponding parameter as declared in the function's prototype. The type of the argument expected is given as `type1`, and the type of the argument actually provided is `type2`. If possible, the argument is converted to the appropriate type as if it were cast to that type. If the argument cannot be converted, an error message is produced.

Warning 89: constant converted from "type1" to "type2"

Your code supplied a constant that conflicted with the expected type. The constant was converted, but the conversion may have caused a loss of precision or lost values. For example, in the following code, the prototype for the function `foo` specifies an `int`, but the function is passed a `double` constant:

```
void foo(int);

void myfunc(void)
{
    foo(10.67); /* Warning 89 */
}
```

The `double` constant is converted to an integer, resulting in an integer argument of 10, and the compiler generates warning 89 to inform you of the conversion.

Error 90: invalid argument type specifier

An error was made in declaring an argument in a function or prototype declaration. For example, the following declaration does not specify a type for the parameter `y`. For example:

```
void foo(int x, y) /* Error 90 */
{
}
```

Error 91: illegal void operand

One of the operands in an expression was of type `void`. The `void` type represents no value and is, therefore, illegal in most expressions.

Warning 92: statement has no effect

An expression statement did not produce an assignment, function call, or other action. Such a statement serves no useful purpose and can be eliminated. This warning is often generated when a typographical error has been made in coding the statement. For example, you might enter

the equality (==) operator when you intended to enter the assignment (=) operator:

```
int i;
i == 5; /* Warning 92 */
```

In this example, the user intended to assign the value 5 to the integer variable `i`, but because of the extra equals sign, the statement does nothing.

Warning 93: no reference to identifier "*name*"

Your code declared an automatic (local) variable but never used the variable. However, the variable might be used by code that has been excluded from the object module with `#if` or `#ifdef` statements.

Warning 94: uninitialized auto variable "*name*"

An automatic (local) variable was used in an expression before it was given a value. Automatic variables are not guaranteed to have any specific value when a function is entered, so an uninitialized automatic variable can create seemingly random bugs. It is possible for this warning to not be produced when appropriate or to be produced incorrectly because the compiler does not check all possible execution paths. In rare cases, the message is produced incorrectly if the variable is used in a loop construct and initialized later in the same loop construct, as shown:

```
void myfunc(void)
{
    int i, j;
    for(i=0; i<10; i++)
    {
        if(i > 0) printf("%d\n", j); /* Warning 94 */
        j = i;
    }
}
```

Without doing detailed loop analysis, the compiler cannot tell that `j` is not referenced by the call to `printf` until after it is initialized.

Some cases that use uninitialized variables may not produce the warning. For example, no warning is produced for the following code even though `j` is not initialized if `x` is greater than 5:

```
void myfunc(int x)
{
    int j;

    if(x<5) j = x;

    if(j < 5) /* j might be uninitialized */
    {
        ...
    }
}
```

Warning 95: unrecognized #pragma operand

The keyword following a `#pragma` statement was not recognized as a SAS/C pragma keyword. Version 6 of the SAS/C Compiler supports the `libcall`, `flibcall`, `syscall`, `tagcall`, and `msg #pragmas`.

Error 96: missing name for #pragma

In a `libcall`, `flibcall`, `syscall`, or `tagcall` `#pragma` statement, the name of the routine to be called was omitted or incorrect.

Error 97: bad library base for #pragma

In a `libcall`, `flibcall`, or `tagcall` `#pragma` statement, the library base was omitted or incorrect.

Error 98: invalid data for #pragma

In a `libcall`, `flibcall`, `syscall`, or `tagcall` `#pragma` statement, the offset or *magic number* fields were invalid. In a `#pragma msg` statement, a syntax error was made specifying the message number or the action to be taken.

Error 99: attempt to change a const lvalue

Your code attempted to modify a variable declared with the `const` keyword. Variables declared with the `const` keyword are read-only and cannot be modified.

Warning 100: no prototype declared for function "name"

Your code called the named function, but the compiler did not find a prototype for that function. Without a prototype, the compiler cannot perform parameter type checking. Remember, a function that takes no parameters still needs a prototype with `void` as its parameter list:

```
int func(void); /* "func" returns "int" and takes no parms */
```

If the function is a SAS/C function, the *SAS/C Development System Library Reference* tells you which header file to `#include` to get the correct definition for the function. If the function is an AmigaDOS function, its prototype is in the `include:clib` directory in the file for its library. For more information, you can refer to the *Amiga ROM Kernel Reference Manual: Libraries, 3rd Edition* (Commodore-Amiga, Inc. 1992) or use the `grep` command to search the header file in the `include:clib` directory for the prototype.

Error 101: redundant keywords in declaration

The same storage class keyword was specified multiple times when declaring a variable, function, or prototype. For example, you may have specified the `const` or `__chip` keyword twice.

However, you can specify a `typedef` with a storage class keyword. Any variables declared with this `typedef` are automatically assigned to that specified storage class. Therefore, you may get an error 101 for specifying the keyword explicitly on a variable declared with the `typedef`, as in the following example:

```
typedef int __far farint;

farint x; /* Declares a far integer called "x" */

farint __far y; /* Error 101 */
```

Error 102: conflicting keywords in declaration

You have specified two mutually incompatible storage class keywords when declaring a variable, function, or prototype. For example, you may have specified the `__near` and `__far` keywords on a variable definition.

However, you can specify a `typedef` with a storage class keyword. Any variables declared with this `typedef` are automatically assigned to that specified storage class. Therefore, you may get an error 102 for

specifying a keyword explicitly on a variable which conflicts with the keyword declared with the `typedef`, as in the following example:

```
typedef int __far farint;

farint x; /* Declares a far integer called "x" */

farint __near y; /* Error 102 */
```

Warning 103: uninitialized constant "[name]"

The named variable was declared `const`, indicating that the variable is read-only, but you have not initialized the variable. If the variable is declared `static` or `extern`, it is initialized to zero for you but is probably not useful unless explicitly initialized.

Warning 104: conversion from pointer to const/volatile to pointer to non-const/volatile

A pointer to a `const` or `volatile` object is being converted to a pointer to a normal object. For example, you may be passing the address of a `const` variable to a function that takes a normal pointer. After the pointer has been converted, the `const` or `volatile` attribute is lost. Therefore, your `const` data might get modified, or your `volatile` data might be kept in a register.

In the following example, warning 104 is produced when the function `foo` is called with the `const` array `data` as an argument.

```
void foo(char *);

void myfunc(void)
{
    const char data[] = "Hello, World!\n";
    foo(data); /* Warning 104 */
    printf("%s\n", data);
}

void foo(char *p)
{
    p[1] = 'a';
}
```

The function `foo` actually modifies its data by writing to it. Therefore, `printf` prints the value `Hallo, World!` instead of `Hello, World!` because the constant array has been modified.

Warning 105: module does not define any externally-known symbols

The C source file and all of its included headers were compiled, but no externally-visible functions or data items were defined. In this case, a valid object file is produced, but it contains nothing that can be accessed, and the file can be removed from your link step if you desire.

Error 106: postfix expression not allowed on a constant

Your code attempted to apply a postfix ++ or – operator to a constant. This message may be generated because of an incorrect macro expansion. You can use the **pponly** option to generate a preprocessed file and check the macro expansion.

Error 107: too many initializers

A data item was being initialized, but too many initializer elements were provided, as in the following example:

```
struct F00
{
    int i, j;
};

struct F00 foo = {10, 20, 30}; /* Error 107 */
```

Warning 108: zero-length arrays are not an ANSI feature

This message is suppressed by default, but you can enable it with the **strict**, **ansi**, or **warn=108** options. You have declared an array of size zero, probably as a member of a structure to allow variable-sized structures. While using zero-length arrays is a common extension to the ANSI Standard and is supported by the SAS/C Compiler, the ANSI Standard does not allow the use of zero-length arrays.

Error 109: invalid use of type name or keyword

A type name (possibly a **typedef**) has been encountered where it was not expected. You may have attempted to declare a variable with the same name as a keyword.

**Warning 110: enum constant expression is wrong type
Expecting "type1", found "type2"**

An enumeration constant value was explicitly assigned that did not match the type of the `enum` being declared. For example, specifying a `double` constant when defining an `enum` or specifying a `long` constant when the `shortint` option is active will generate this message.

Example:

```
enum COLORS { RED=1.0 }; /* Warning 110 */
```

Warning 111: non-portable enum type specified

This message is suppressed by default, but you can enable it with the `strict`, `ansi`, or `warn=111` options. As an extension to the ANSI Standard, the SAS/C Compiler supports `short`, `char`, and `long` `enum` types with the syntax `short enum`, `char enum`, and `long enum`. To improve the portability of your code, enable this warning.

**Warning 114: negative shift or shift too big for type
shifts for type "type" must be between 0 and number
bits**

Your code attempted to shift an integral value by a constant number of bits, but the constant was either negative or higher than the number of bits in the value being shifted, as shown in the following examples:

```
int i = 1;
i = i >> 33; /* Warning 114 */
i = i << -1; /* Warning 114 */
```

In the first example, `i` is 32 bits (an `int`) but is being shifted 33 bits. Shifting a value by a number of bits higher than the number of bits in the value generates a zero for the result. In the second example, `i` is being shifted by a negative number of bits. Attempting to shift a value by a negative number of bits usually generates a zero for the result (although this result is undefined by the ANSI Standard).

**Error 115: enum constant value "number" out of range for enum
type**

Your code specified a value for an `enum` constant while defining an enumerated type which is out of range for the type of `enum` being declared. Remember that `enum` constants that are not explicitly assigned a numerical value are given the value of the previous constant defined

in the same `enum` incremented by one. This increment might place the constant's value out of the acceptable range, as in the following example:

```
char enum COLORS
{
    RED=254,    /* Numeric value is 254          */
    GREEN,     /* Numeric value is 255 (254+1) */
    BLUE       /* Numeric value is 256 - too big for char */
};             /* Error 115 is issued */
```

The `enum` constant `RED` is assigned a value of 254. `GREEN` does not have an explicit value, so it is assigned 254+1, or 255. `BLUE` does not have an explicit value, so its value would normally be 255+1, or 256. However, the `enum` is declared as being a `char enum`, so 256 is too big and error 115 is issued.

Warning 116: undefined enum tag "name"

A reference was made to an `enum` that has not yet been declared. The compiler may generate this message as a warning if only a pointer to the `enum` is used, but the compiler generates this message as an error if any other use is made of the `enum`.

Error 117: enum contains no members

An attempt was made to define an enumerated type, but no member names were specified. Use the `pponly` option to generate a preprocessed file if you have problems eliminating this message.

Error 118: conflicting use of enum/struct/union tag "name"

An attempt was made to define an `enum`, `struct`, or `union` that has the same name as a previously defined `enum`, `struct`, or `union` in the same scope. You cannot define an `enum` with the same name as a `struct`, `union`, or another `enum`.

**Error 119: identifiers missing from definition of function
"name"**

A prototype-style function definition was encountered, but the names of the parameters were missing. You can omit the parameter names from a prototype, but the names must be present in the actual function definition. For example the following code generates error 119:

```
void func(int, int, double); /* Legal prototype */

void func(int, int, double) /* Illegal function definition */
{                             /* Error 119 */
}
```

**Warning 120: Integral type mismatch: possible portability
problem
Expecting "type1", found "type2"**

This message is suppressed by default, but you can enable it with the `strict` or `warn=120` options. The wrong integral type was passed as a parameter. This error may be a problem on another compiler or another type of computer if the size of an `int` is different from the size of an `int` on the Amiga computer. This warning is not a problem if your code needs to run with the SAS/C Compiler only.

In the following example, the SAS/C Compiler promotes the short integer to a long integer even if you compile with the `shortint` option:

```
#pragma msg 120 warn
void func(long);

void main(void)
{
    short s = 10;
    func(s);      /* Warning 120 */
}
```

However, pre-ANSI compilers on machines that define the `int` type to be the same as `short` may not perform the conversion or may perform the conversion and issue a warning message. In addition, if you remove the prototype, this code will not work on any machine where `int` is the same as `short`.

**Warning 121: hex/octal constant "constant" too large for char
High bits may be lost**

Warning 121 is generated if a hexadecimal or octal constant specifies more digits than will fit into a single `char`.

According to the ANSI Standard, hexadecimal bytes may be specified in a quoted string using the `\xhh` escape sequence. The ANSI Standard also states that the escape sequence consumes all valid hexadecimal characters in the string following the `\x`, even if all the characters consumed will not fit into a single byte, which may not be what you want. For example, in the case of `\xabcdefgh`, you may want the character `0xab` first, followed by the string `cdefgh`. According to the ANSI Standard, this escape sequence evaluates to the character `0xef` followed by the string `gh`, because the `\x` consumes all valid hexadecimal characters. A single byte cannot hold the hexadecimal number `0xabcdef`, so the top bits are lost and `0xef` is the result. The best way to get the character `0xab` followed by the string `cdefgh` is to use ANSI string concatenation to terminate the escape sequence: `\xab cdefgh`. The two strings are concatenated to form the desired sequence.

Warning 122: missing ellipsis

A function or function prototype was encountered that attempted to specify variable arguments with a trailing comma in the argument list. According to the ANSI Standard, variable arguments must be specified by a trailing comma followed by an ellipsis (...). The SAS/C Compiler accepts the form without the ellipsis as if the ellipsis had been specified, but other ANSI-conforming compilers may require the ellipsis, as in the following example:

```
void foo(int x, ... ); /* Right */
void bar(int x, );    /* Wrong */
```

**Warning 123: no tag defined for enumeration
Cannot construct prototype**

The `genproto` option was active, but a prototype could not be generated for a function because an `enum` was used as a parameter to the function and that `enum` had no tag, as shown in the following example.

```
void myfunc(colors)
enum {RED, GREEN, BLUE} colors;
{
}
```

The *tag* is the name normally appearing immediately after the keyword **enum** in the **enum** declaration. This message is produced only if the **genproto** compiler option is used to produce prototypes.

Error 125: invalid number

The numerical constant specified cannot be represented on the Amiga computer. The constant is probably an integer that is too large or too small (negative) to be represented in four bytes. This error can also be produced if a floating-point number is specified incorrectly or if invalid digits are provided to an octal constant.

The following lines all produce error 125:

```
double d = 10.3.2;           /* Error 125 */
int i = 123456789123456789; /* Error 125 */
int j = 099;                 /* Error 125 */
```

In the first line, the value assigned to the floating-point variable has been incorrectly specified. In the second line, the value contains too many digits to fit into an **int**. In the third line, the value begins with a zero, which makes the value an octal constant, but it contains the digit 9, which is invalid for octal constants.

Warning 126: #endif, #else, or #elif out of order

A **#endif**, **#else**, or **#elif** was encountered but no **#if** or **#ifdef** was active.

Error 127: operand to # operator must be a macro argument

If a preprocessor macro parameter is preceded with the **#** operator, that parameter is replaced with the literal string consisting of the corresponding argument to the macro. For example, if you pass the argument **FOO** to a macro, and that argument in the macro definition is preceded by the **#** operator, that argument expands to the string **"FOO"**. The **#** operator can be applied only to macro arguments.

Error 128: text-from-#error

This message is the result of a **#error** preprocessor directive in the source code being compiled. The text of the message is taken from the **#error** line.

Error 129: ambiguous struct or union member "name"

Your code referred to a structure or union member that did not exist. In an attempt to resolve the name, the compiler examined the names of all members of substructures or subunions. Two or more members of subaggregates matched the name you provided, so the compiler was unable to determine which member you intended to specify.

Suppose you have the following code:

```
void main(void)
{
    struct FOO
    {
        int x, y, z;
    };

    struct BAR
    {
        int a, b, x;
    };

    struct COMBO
    {
        struct FOO foo;
        struct BAR bar;
    } combo;

    combo.a = 10; /* Warning 193 */
    combo.x = 10; /* Error 129 */
}
```

The reference to `combo.a` succeeds, although it generates a warning 193 (**implicit reference to structure member**), because the full reference should be `combo.bar.a`. The reference to `combo.x` generates error 129, because both substructures of `combo` contain a member called `x`. The compiler cannot tell whether you mean `combo.foo.x` or `combo.bar.x`, so it issues the error message 129.

See Chapter 11, “Using Amiga Specific Features of the SAS/C Language,” for more information on implicit structure references.

Error 131: maximum temporary or formal storage exceeded

The compiler must allocate stack storage to allow you to pass and receive structures or other parameters to and from functions. Parameters are copied onto the stack for each function call, so if you are passing large structures, your code will run very slowly. Consider passing a pointer to your structure instead of the entire structure.

Warning 132: extra tokens after valid preprocessor directive

The ANSI Standard does not allow extra text on a preprocessor line, but this extra text may be allowed by some C compilers. For example, many programmers put a descriptive word after a `#endif` directive indicating the `#if` to which the `#endif` belongs. To make your code ANSI-compatible, you should place such descriptive words in comments, as shown in the following example:

```
#ifdef SOMETHING
#endif SOMETHING /* Warning 132 */

#ifdef SOMETHING
#endif /* SOMETHING */ /* No warning */
```

Error 133: cannot redefine macro "name"

An attempt was made to redefine a built-in preprocessor macro, like `__FILE__` or `__LINE__`. You cannot redefine built-in preprocessor macros.

Warning 134: too many arguments

A macro definition was encountered with more than the allowable number of arguments. The current ANSI limit for macro arguments is 31, and the SAS/C Compiler does not allow more than 31 arguments. If message 134 is issued, the macro is not successfully defined. Any use of the macro in your code produces a warning for a function with no prototype and possibly a linker error when the name of the macro cannot be resolved by the linker.

**Error 135: argument count incorrect for macro "name",
expecting *number* arguments
See line *number* file "*filename*"**

Your code invoked a macro with too few or too many arguments. The macro is defined at the specified file and line number.

Error 136: invalid use of register keyword

Your code used one of the register keywords inappropriately. The register keywords are `register`, `__a0`, `__d0`, `__a1`, `__d1`, and so on.

Warning 137: ANSI limits #line numbers to between 1 and 32767

This message is suppressed by default, but you can enable it with the **strict**, **ansi**, or **warn=137** options. The ANSI Standard states that line numbers specified with a **#line** directive should be between 1 and 32767. However, the SAS/C Compiler accepts any number that will fit into a signed four-byte integer.

Error 138: operation invalid for pointer to void

Your code attempted to perform an operation on a **void *** pointer that is not valid for **void ***. Such operations include addition, subtraction, array indexing, and dereferencing (unary ***** operator). For example, the following code attempts to perform addition on a **void *** pointer:

```
void *p = NULL;

p = p + 1; /* Error 138 */
```

Warning 139: missing #endif
See line number file "filename"

The compiler encountered an **#if**, **#ifdef**, or **#elif** statement for which there is no matching **#endif**. The file and line number of the unmatched directive is specified in the message. This message occurs only at the end of the source file, so large portions of your program may have been ignored because of a stray **#if** or **#ifndef** with no matching **#endif**.

Warning 140: sizeof operator used on array that has been converted to pointer

Use of the **sizeof** operator on an array name gives the size of the array. However, if the array name has been converted (cast) to a pointer, the resulting size is the size of the pointer (4 bytes), not the size of the array. Almost any use of an array name in an expression implicitly casts the array name to pointer. To determine the actual size of the array, pass only the array name to the **sizeof** operator.

In the following example, in the first assignment statement, `i` gets the value 4 (the size of a pointer to a character). In the second assignment statement, `i` gets the value 10.

```
char ary[10];
int i;

i = sizeof(ary+1); /* Warning 140 */
i = sizeof(ary);
```

Error 142: array size never given for "name"

You can declare an array as `extern` and not specify the first subscript. However, when you define the array, you must specify the subscript. Storage is allocated when you define an array, not when you declare an array as `extern`. For example:

```
extern char ary1[][10]; /* Legal */

char ary2[][10];      /* Error 142 */
```

Error 143: object has no address

An attempt was made to take the address of something that has no address. You may be attempting to take the address of a register variable or an expression.

Warning 146: case value out of range for switch type

A `case` value was specified in a `switch` statement that can never be taken. For example, if you compile with the `shortint` option and a `switch` statement is switching on a `short` or `int` value, `case` values too big to be stored in 16 bits will generate this warning:

```
/* The SHORTINT compiler option is on. */
int i = 10;

switch(i) /* i is a 16-bit integer */
{
    case 0x08000000: /* Warning 146 */
        break;

    case 65000:      /* Warning 146 */
        break;
}
```

Warning 147: conversion between function and data pointers

Your code has cast or otherwise converted a pointer that points to a function to a pointer that points to data. The ANSI Standard states that this conversion is not legal. This conversion works with the SAS/C Compiler in most circumstances, but you should use the **absfuncpointer** compiler option if your code is larger than 32K.

**Warning 148: use of incomplete struct/union/enum tag "*name*"
See line *number* file *filename***

This message is suppressed by default, but you can enable it with the **warn=148** option. An incomplete tag is one for which no definition has been seen. The ANSI Standard allows use of an incomplete tag (for declaring pointers, for example). If you want to know when an incomplete tag is being used, enable this warning. You may also want to enable warning 149.

Warning 149: incomplete struct/union/enum tag in prototype scope "*name*"

This message is suppressed by default, but you can enable it with the **strict** or **warn=149** options. An incomplete tag was discovered in a prototype. An incomplete tag is one for which no definition has been seen. The ANSI Standard requires the incomplete tag's scope to end at the end of the prototype. Therefore, any definition of the function later, even if the tag in question has been defined, may generate an error message and terminate the compilation. You may want to turn on warning 148 for the most information about incomplete tags. If you do not want to see all the incomplete tag information, you identify most problem cases by enabling only warning 149.

The following function definition produces an error on many ANSI-conforming compilers because the structure **FOO** referred to in the definition is considered to be a different structure **FOO** than the one referred to in the prototype.

```

void func(struct FOO *); /* Warning 149 */

struct FOO
{
    int x, y, z;
};

/* Many compilers issue an error here. */
void func(struct FOO *foo)
{
}

```

This error could be fixed by moving the definition for the structure **FOO** before the prototype.

Warning 150: the keyword "name" is meaningless for itemtype

When declaring a function or data item, you have used a keyword that is meaningless for that type of item. For example, you cannot specify the following:

```
void __chip foo(int); /* Warning 150 */
```

It is meaningless to say that a function is to be allocated in **__chip** memory.

The **__near** and **__far** storage class keywords are valid on both functions and data items, but have slightly different meanings. See the descriptions of the **data=near** and **code=near** options in Chapter 8, "Compiling and Linking Your Program," for more information.

Some keywords that are valid on functions are also valid on data items because the data items may be pointers to functions of the designated type. For example, you may have a function that returns a pointer to a **__regargs** function. Keywords in this category include **__stdargs**, **__regargs**, **__asm**, and **__interrupt**.

Other keywords need to be present only on the function definition. You do not need to add them to function pointers and function prototypes. Keywords in this category include **__stackext** and **__saveds**. If these keywords appear on a data item, warning 150 is generated and the extra keyword is ignored.

For more information on specifying keywords, see Chapter 11, "Using Amiga Specific Features of the SAS/C Language."

Error 152: cannot define function via typedef name

Your code attempted to define a function using a **typedef**, as shown below:

```
typedef int foo(int);

foo bar /* Error 152 */
{
}
```

According to the ANSI Standard, you cannot define a function using a **typedef**. The SAS/C Compiler does not accept such a definition.

Warning 154: no prototype declared for function pointer

Function pointers require prototypes just like functions. To declare a prototype for a function pointer, enter the parameter list instead of the empty parentheses in the definition, as shown in the following example:

```
void (*func1)(); /* Warning 154 */

void (*func2)(int, double, long); /* No warning */
```

Remember that a pointer to a function that takes no parameters still requires a prototype:

```
void (*func3)(void); /* Function pointer takes no parms */
```

Warning 155: no statement after label

The C language does not allow a label immediately before the curly brace ending a block (`{}`). If you enter a label in this position, the SAS/C Compiler generates this warning. Enter a semicolon (representing a **NULL** statement) after the label to suppress the warning, as shown in the following example:

```
void func1(void)
{
    goto foobar;
    /* more code */
    foobar:    /* Warning 155 */
}

void func2(void)
{
    goto foobar;
    /* more code */
    foobar: ;  /* No warning */
}
```

Warning 156: operation/comparison of pointer to "int" and pointer to "type"

This message is suppressed by default, but you can enable it with the **warn=156** options. If you compiled with the **shortint** option, your program has used a pointer to **int** and a pointer to **short** in an operation. If you did not compile with the **shortint** option, your program has used a pointer to **int** and a pointer to **long** in an operation.

For example, the following code will work if `int` and `long` are the same length, but the same code will not compile if `ints` and `longs` are different lengths:

```
/* Assume SHORTINT is not active */
int *ip = NULL;
long *lp = NULL;
int diff;

diff = ip - lp;
```

Error 158: invalid type name

The compiler expected a type name for a cast or `offsetof` operation, but the provided name was not recognized as a valid type name. Check the preprocessed output to make sure you are providing the correct information.

Warning 159: use of unary minus on unsigned value

This message is suppressed by default, but you can enable it with the `strict` or `warn=159` options. Your code has used the unary minus operator (`—`) on an unsigned variable. Using this operator on an unsigned variable may not produce the expected result because the result is still a non-negative number.

Warning 161: no prototype declared at definition for function "name"

An old-style function definition was encountered, and no prototype was in scope. The prototype must appear in the C file or a header file before the definition of the function, or the function definition itself must be a prototype-style definition.

Warning 162: non-ANSI use of ellipsis punctuator

This message is suppressed by default, but you can enable it with the `strict`, `ansi`, or `warn=162` options. You have declared a function with the ellipsis punctuator (`...`) but the function has no arguments. The ANSI Standard requires functions that take a variable number of arguments to take at least one fixed argument.

Warning 163: initialization of auto struct, union, or array

This message is suppressed by default, but you can enable it with the `strict` or `warn=163` options. Your code has initialized an automatic structure, union, or array. The ANSI Standard allows you to initialize automatic structures, union, and arrays, but many pre-ANSI compilers do not.

Warning 164: & applied to array

This message is suppressed by default, but you can enable it with the **strict** or **warn=164** options. Your code has used the address (&) operator on an array name. You should instead take the address of the first element in the array, or use the array name without the address operator. For example:

```
int ary[10];
int *iptr;

iptr = &ary;          /* warning 165 */
iptr = ary;           /* OK          */
iptr = &ary[0];       /* OK          */
```

Warning 165: use of narrow type in prototype

This message is suppressed by default, but you can enable it with the **warn=165** option. This warning is provided for detecting situations that may cause problems on other compilers if your code mixes function prototypes and old-style function definitions. See Chapter 13, “Writing Portable Code,” for information on using narrow types in function declarations. See also the descriptions of messages 176 and 179.

**Error 166: unrecoverable error or too many errors
Terminating compilation**

Your program has exceeded the default or specified **maxerr** or **maxwarn** values, or an error has occurred that prevents the compiler from producing meaningful results.

The default maximum number of errors is 50. By default, any number of warnings may be generated.

**Warning 169: incompatible operands of conditional operator (?:)
"type1" conflicts with "type2"**

The operands of the **?:** conditional operator must be of compatible types. Your code supplied types to the **?:** operator that were not compatible. For example, the following code generates warning 169:

```
int func(void)
{
    int i = 0;
    struct FOO *foo = NULL;

    return (int)(i > 0 ? foo : i); /* Warning 169 */
}
```

The expression following the question mark (?) is of type `struct FOO *`. The expression following the colon (:) is of type `int`.

This message can be an error if it is not possible to convert the types in question. This can occur if one of the types is a structure or union.

Warning 170: overflow during operation on constants

A constant expression overflowed the limits of the type in which it was being calculated. Perhaps you have added or multiplied two large integers, thereby resulting in a number too large to represent in a four-byte integer.

Warning 176: implicitly promoted formal "name" conflicts with prototype

See line *number* file "*filename*"

This message is suppressed by default, but you can enable it with the `strict`, `ansi`, or `warn=176` options. You are defining a function using an old-style definition, which means that any narrow types (`char`, `short`, and `float`) in your definition are implicitly widened to their non-narrow equivalents (`int`, `int`, and `double`, respectively). However, a prototype is in scope that gives the narrow version of the type.

Functions that call your function will not know that your function is using an old-style definition and, with some compilers, may pass an incorrect value. An incorrect value is never passed using the SAS/C Compiler on the Amiga hardware. See Chapter 13, "Writing Portable Code," for information on using narrow types in function declarations. See also the descriptions of messages 165 and 179.

Warning 178: indirect call without indirection operator

This message is suppressed by default, but you can enable it with the `strict` or `warn=178` options. You called a function using a function pointer, but did not use the indirection (*) operator to dereference the pointer first. The SAS/C Compiler generated correct code, but on pre-ANSI compilers this code may not work. Example:

```
void (*funcptr)(int);

funcptr(10); /* Warning 178 */
(*funcptr)(10); /* No warning */
```

Warning 179: narrow type used in old-style definition

This message is suppressed by default, but you can enable it with the **strict** or **warn=179** options. Your code has used a narrow type (**char**, **short**, or **float**) in an old-style definition. See Chapter 13, "Writing Portable Code," for information on using narrow types in function declarations. See also the descriptions of messages 165 and 176.

Warning 180: no space between macro name and its replacement list

This message is suppressed by default, but you can enable it with the **strict**, **ansi**, or **warn=180** options. Your code is defining a preprocessor macro that takes arguments but does not have at least one blank after the closing parentheses of the argument list. The ANSI Standard requires white space after the closing parentheses. For example:

```
#pragma msg 180 warn
#define ADD(a,b)(a+b) /* Warning 180 */
```

**Warning 181: "name" was declared both static and external
See line number file "filename"**

Your code has declared a function as both **static** and **external** at different places. Both the function and its prototype must agree on whether the function is static.

**Warning 182: static function "name" declared but not defined
See line number file "filename"**

Your code had a prototype for a function declared **static** but never defined the function. The function definition may be hidden with **#if** or **#ifdef** statements.

**Warning 183: inline function declared but not defined
See line number file "filename"**

Your code had a prototype for a function declared **__inline** but never defined the function. The function definition may be hidden with **#if** or **#ifdef** statements.

Warning 184: invalid multi-byte character constant

Your code has specified a multibyte character constant (such as **'ab'**), and you did not compile your code with the **mccons** option; or, if you compiled your code with the **mccons** option, the multibyte character is larger than the permitted four bytes.

Error 185: comma expected

The compiler expected a comma but did not find one. This error may be produced because of errors in previous statements. Fix all previous errors before fixing this one.

Warning 186: implicit conversion between pointer and scalar

Your code has converted a pointer to an integer while doing static initialization. You can suppress this warning by casting the pointer to the appropriate type.

Warning 187: negative value assigned to unsigned type

This message is suppressed by default, but you can enable it with the `strict` or `warn=187` options. Your code has assigned a negative constant to an unsigned variable. In doing so, your code is in effect assigning a very large positive number to the variable, which may or may not be the action you intended.

**Warning 190: #include ignored because header already included
See line *number* file "*filename*"**

This message is suppressed by default, but you can enable it with the `warn=190` option. The `nomultipleincludes` compiler option is active, and your code included the same header file more than once. The compiler ignores the additional `#include` statement. You can use this message to locate all such multiple includes if you want to modify your header files to not include the same file more than once.

You can get even more information about which header files were included by using the `listincludes` compiler option.

Warning 192: wrong size for enum

An `enum` variable was declared to be of a different type than the base `enum` was when it was defined. Suppose you have the following code:

```
char enum COLORS = {RED, GREEN, BLUE};
enum COLORS color; /* Warning 192 */
char enum COLORS color2; /* correct */
```

The `enum` variable `color` should be declared the same size as the base `enum` type, which is `char enum`. The base `enum` type overrides the `enum` variable's type definition.

**Warning 193: implicit reference to struct/union member
Reference assumed to be "reference"**

Your code has referred to a **struct** or **union** member name that is actually a member of a substructure or subunion of the original. The compiler has searched all substructures and subunions and determined that exactly one member matches the name you specified, so it is using that member. The **reference** printed in the message text is the fully expanded name of the member that it is using. If you intend to take advantage of this feature of the SAS/C Compiler, you may disable this warning with the **ignore=193** option. If you choose to disable this warning, your code may not work on other compilers.

For more information on using implicit structure references, see Chapter 11, "Using Amiga Specific Features of the SAS/C Language."

**Warning 194: too much local data for NEAR reference,
some changed to FAR**

You have declared more than 32K of **static** data. The compiler can address up to this amount using 16-bit offsets. Amounts greater than 32K must be addressed using 32-bit offsets. You may have compiled with the **data=faronly** option and declared some data with the **__near** keyword. If your entire project is in one source file or is compiled with **data=faronly**, you can ignore this warning unless you get an error later in the compilation or link. Otherwise, you must eliminate some near data in one of three ways:

- Compile your program with the **strmerge** compiler option. This option moves all string constants to the code section, thereby moving them out of the near data section.
- Declare read-only data as **static const**. When you compile with the **stringmerge** option, this data will also be moved to the code section.
- Add the **__far** keyword to some of your larger external or static data items to move those items from the near section to the far section.
- Specify the **data=far** compiler option to move all data items except those declared with the **__near** keyword to the far section.

Note: The second line of message 194 is not printed unless you compiled with the **data=auto** option, or you compiled with the **data=faronly** option and declared data with the **__near** keyword.

Warning 195: nested comment detected

This message is suppressed by default, but you can enable it with the `warn=195` option. The compiler found the start of a comment (`/*`) inside of a comment. Some compilers allow comment nesting and others ignore the start of the second comment. The ANSI Standard does not allow nested comments. If you choose to use nested comments, you should specify the `comnest` compiler option. However, using the `comnest` option prevents your code from running on most ANSI-compliant compilers.

Warning 196: specified include directory not found: "name"

The named directory does not exist or could not be accessed.

Warning 198: __regargs invalid for a varargs function

Functions that take variable numbers of parameters always pass their parameters on the stack. The `__regargs` keyword is invalid for these functions.

Error 199: unbalanced comment

See line *number* file *filename*

Your code has a comment open (`/*`) with no corresponding comment close (`*/`). This condition is not detected until the end of the C source file.

Warning 204: macro invocation not terminated

Your code invoked a macro but did not supply enough closing parentheses to terminate the invocation. Also, check any of the macros that are invoked by the macro that you invoked directly in your code.

Warning 209: macro invocation may have multiple side effects

You have passed an expression with a side effect as an argument to a macro, and that macro has evaluated the argument more than once. Operators with side effects are `++`, `--`, `+=`, `/=`, `*=`, `-=`, `>>=`, `<<=`, `|=`, `&=`, and `=`.

For example, the `max` and `min` macros evaluate each argument twice. If you invoke the `max` macro as `MAX(i++, j)`, the first argument is evaluated twice, and two post-increments take place. You could define the `max` macro as follows:

```
#define MAX(x,y) ((x) > (y) ? x : y)

int i = 0;
i = MAX(i++, 0);    /* Warning 209 */
```

The macro expands to:

```
i = ((i++) > (0) ? i++ : 0);
```

The expansion contains the expression `i++` twice, so if that branch of the `?:` operator is executed, the variable `i` is incremented twice.

This warning may sometimes be produced incorrectly, but you should examine each case to determine whether there is an error.

Function calls are also considered side effects, but function calls produce warning 217 instead of warning 209.

Warning 212: item "*name*" already declared
See line *number* file "*filename*"

This message is suppressed by default, but you can enable it with the `warn=212` option. You have declared an item twice. For example, you may have specified the prototype for a function twice, or you may have declared an `extern` twice. The declarations do not conflict, or the compiler would generate error 72, but code is harder to maintain if it defines the same item in more than one place. You may want to use the `nomultipleincludes` option to suppress multiple `#includes` of the same file if you enable warning 212.

Warning 213: empty argument to preprocessor macro

This message is suppressed by default, but you can enable it with the `strict`, `ansi`, or `warn=213` options. Your code calls a preprocessor macro with no argument text. This action is accepted by the SAS/C Compiler but is not permitted by the ANSI Standard. For example:

```
#define F00(a,b)  a

F00(10,)
```

Warning 216: symbol "*name*" found

You have compiled with the `fsym` compiler option, and the compiler has found a definition of one of the symbols that you specified as the parameter to the `fsym` option.

Warning 217: macro invocation may call function multiple times

You have passed a function call as an argument to a macro that evaluates its arguments more than once. This action may cause incorrect results. See also the description of message 209.

Error 218: declaration found in statement block

The compiler found a declaration where it expected a statement. You may have a statement in the middle of your declaration block, or vice-versa. Any declarations found after this error are added as external variables, which suppresses warnings about undefined variables but may create additional errors later if your code declares a variable of the same name. In the following example, the variable `y` is declared after the first statement of the function:

```
void func(void)
{
    int x;
    x = 10;
    int y;  /* Error 218 */
}
```

Warning 220: old-fashioned assignment operator taken as "operators"

This message is suppressed by default, but you can enable it with the `strict` or `warn=220` options. Older compilers allowed the operators `-=`, `+=`, and so on. In ANSI C, these operators are specified as `-=`, `+=`, and so on. The older specification is ambiguous when assigning certain expressions (for example, `x=-5;` and `x= -5;`). To resolve this ambiguity, newer compilers use `-=`, and some pre-ANSI compilers use `-=`. Therefore, you should enter at least one space between the equals (`=`) sign and whatever operator follows it to ensure portability.

Error 223: "filename" is not a valid GST file

The specified filename was supplied using the `gst` compiler option as a Global Symbol Table file, but it is not a valid GST file. Delete the bad file and try re-creating it.

**Warning 224: item "name" already defined
See line number file "filename"**

This message indicates that a variable, function, or `typedef` is being defined for the second time in a given scope. This message is normally an error, but if the redefinition is harmless, the compiler allows it and issues the message as a warning instead. As shown in the following example, the compiler allows redefinition of `typedef` names in the same scope if the new type is identical to the old type. The compiler does not allow redefinition of functions or variables in the same scope.

```

typedef int foo;
typedef int foo; /* Warning 224 */

void func(void)
{
}

void func(void) /* Error 224 */
{
}

```

Warning 225: pointer type mismatch
"type1" does not match "type2"

Your code has performed an assignment or some other operation on two incompatible different pointer types or on a pointer type and an arithmetic type. For example:

```

struct F00
{
    int x, y, z;
} foo;
int *ip;

ip = &foo; /* Warning 225 */

```

Error 226: cannot convert "type1" to "type2"

The compiler was unable to perform a requested or implicit conversion. For example, if one of the types is an instance of a structure, the structure cannot be cast to another structure type or to an arithmetic type. For example:

```

struct F00
{
    int x, y, z;
} foo;
int i;

i = (int) foo; /* Error 226 */

```

Explanations of Linker Messages

Error 103: Out of memory!!

slink has run out of memory. By default, **slink** attempts to cache the object modules in memory between pass 1 and pass 2. If **slink** runs out of memory, it attempts to free the cached object modules. In some cases, **slink** cannot find enough continuous memory. Try adding **bufsize 4096** to the **slink** command. This option turns off object module caching.

Error 425: Cannot find library *library-name*

The linker cannot find the specified library. Usually, SAS/C libraries are located in the **LIB:** directory. You may have misspelled the library name in your **slink** command.

Error 426: Cannot find object *name*

The linker cannot find the specified object.

Error 443: *filename* is an invalid file name

The linker found an invalid character in a filename.

Error 444: Hunk_Symbol has bad *symbol-type* symbol *symbol-name*

The object module is corrupt. Try recompiling the object module.

Error 445: Invalid HUNK_SYMBOL *symbol-name*

The object module is corrupt. Try recompiling the object module.

Error 446: Invalid symbol type *symbol-type* for *symbol-name*

Either the object module is corrupt, or **slink** has found a symbol type that it does not recognize. Try recompiling the object module. For a list of valid symbol types, refer to the description of object file structure in *The AmigaDOS Manual, 3rd Edition*.

Error 447: *filename* is a load file

The file specified is an executable module instead of an object module.

Error 448: *filename* is not a valid object file

The file specified is not an object module. The file may be a C source file or a corrupt object module.

Error 449: No hunk_end seen for *filename*

The object module is corrupt. Try recompiling the object module.

Error 450: Object file *filename* is an extended library

A library file was specified as an object module. Add the **library** keyword in front of the library name.

Error 501: Invalid Reloc 8 or 16 reference

An 8- or 16-bit relocation address cannot reach its destination. The SAS/C Compiler does not generate 8-bit relocation records, but third party products may use them. The object module is probably corrupt. Try recompiling your source file. If recompiling the file does not correct the error, contact the Technical Support Division.

Error 502: *function-name* symbol - Distance for Reloc16 greater than 32768

The distance from the point where the function is called to the function itself is greater than 32767 bytes. The function cannot be referenced with a 16-bit address field. Normally, `slink` tries to insert an ALV (Automatic Link Vector) at the end of the calling module, but if the module has more than 32K of code, the relocation may not reach the ALV. An ALV is the instruction used to reach functions that would otherwise be too far away. Compile the file with `-code=far`, or declare the function with the `__far` keyword.

Error 503: *function-name* symbol - Distance for Reloc8 greater than 128

The distance from the point where the function is called to the function itself is more than 128 and so the function cannot be referenced with an 8-bit address field. The SAS/C Compiler does not generate 8-bit relocation records, but third party products may use them.

Error 504: *variable-name* symbol - Distance for Data Reloc 16 greater than 32768

The distance for the 16-bit relocation is too far. You may see this message if a data item (such as a structure or an array) is larger than 64K or if the total amount of data in your program is greater than 64K. To correct the problem, you can either break the data item down and make it smaller than 64K, or you can place the `__far` keyword on the definitions of one or more data items to force all references to those items to be 32-bits. Using the `__far` keyword reduces the amount of data in the near data section to less than 64K. You can also compile your program with the `data=far` option. However, using `data=far` increases code size and execution time.

Error 505: *variable-name* symbol - Distance for Data Reloc8 greater than 128

The distance for the 8-bit relocation is too far. The SAS/C Compiler does not generate 8-bit relocation, but third party products may use them.

Error 506: Can't locate resolved symbol *symbol-name*

If this message appears, please contact the Technical Support Division. See Chapter 3, "Getting Help," for a complete list of items that you need to provide to the Technical Support staff.

Error 507: Unknown Symbol type *symbol-type* , for symbol *symbol-name*

Either the object module is corrupt, or a `slink` has found a symbol type that it does not recognize. Try recompiling your source file. For a list of valid symbol types, see the description of object file structure in *The AmigaDOS Manual, 3rd Edition*.

Error 508: Symbol type *symbol-type* unimplemented

Either the object module is corrupt, or a `slink` has found a symbol type that it does not recognize. Try recompiling your source file. For a list of valid symbol types, see the description of object file structure in *The AmigaDOS Manual, 3rd Edition*.

Error 509: Unknown hunk type *symbol-type* in Pass2

If this message appears, please contact the Technical Support Division. See Chapter 3, "Getting Help," for a complete list of items that you need to provide to the Technical Support staff.

Error 510: *symbol-name* symbol - Near reference to a data item not in near data section

The linker expected the specified symbol to be in the near data section, but the symbol is located in either the far data section, chip memory, or the code section.

You may have defined a variable as `__far` with the `__far` keyword in one module and externally declared the same variable in another module without the `__far` keyword. If so, the module with the external declaration attempted to reference the data as near. To correct the problem, the definition and all declarations of the data item must be specified with the same keywords.

If you compiled your file with the `data=auto` option, the compiler can place variables into the far data section if necessary. The compiler may have placed a variable in one module in the far section and a variable in another module in the near section. To correct this situation, either stop using the `data=auto` option, and use the `__far` keyword (if necessary) on the definition and all declarations of the variable that causes the error.

Error 512: Invalid branch to *function-name* in overlay node *module-name*

The linker detected a branch in an overlay node that calls another overlay node at the same level. This type of branch is not supported by the overlay manager.

Error 513: Multiple NTRYHUNK segments not permitted

The NTRYHUNK is the root overlay node. No other hunks should have this name.

Error 514: Overlay manager `_ovlyMgr` is undefined

This `_ovlyMgr` function is located in SAS/C libraries. To use the SAS/C overlay manager, link with `sc.lib` or `scs.lib`. You can also write your own overlay manager. See Chapter 8, "Compiling and Linking Your Program," for information on creating your own overlay manager.

Error 515: An ALV was generated pointing to data *variable-name* symbol

A 16-bit relocation could not reach its destination, so `slink` inserted an ALV (Automatic Link Vector) instruction, and then determined that the destination is not in a code hunk. To correct the problem, make the reference a 32-bit reference. This error may be generated for object code produced by third-party assemblers and compilers

Error 516: Attempt to merge BSS with CODE or CODE/DATA

Either you have attempted to merge the far BSS section with the far data section, or you have attempted to merge the near or far BSS section with the code section. You can merge a BSS section with the `__MERGED` section only. You will see this message if you have assigned the same name to either the far BSS section and the far data section or the far BSS section and the code section.

If you get this error in another situation, please call the Technical Support Division.

Note: Do not name the code section `__MERGED`.

Error 600: Invalid command *command*

The option listed is not a valid `slink` option.

Error 601: *option* option specified more than once

The option has been specified twice.

Error 602: Unable to open output file *filename*

slink cannot open the output file for write access. Another program such as CodeProbe may have left a lock on the file. Alternatively, the protection bits may prohibit write access to the file, or that the name specified may be a directory.

Error 603: *string* is not a valid number

The linker found a non-numerical character in a field where it expected to find a number.

Error 604: with file is not readable

The with file may contain binary characters or may be locked.

Error 605: Cannot open with file *filename*

The with file does not exist or may be locked.

Error 607: No FROM/ROOT files specified

You did not specify an object module, or if you are using overlays, the root node did not contain any objects.

Error 608: Premature EOF encountered

The object module is corrupt. Try recompiling the source file.

Error 609: Error seeking in file *filename*

The object module is corrupt. Try recompiling the source file.

Error 610: *module-name* has no parent in overlay tree

The with file specifies an overlay with no parent node.

Error 611: Reloc found with odd address for symbol *symbol-name*, file *filename*

The linker has found a relocation record with an odd address. This message can only be generated if your program is written in assembler and is usually caused by placing an odd length character string in the code section.

**Error 612: MERGED Data relocation to non-code section in Overlay Node
Reference at offset *hex-address* in *module-name*, To Unit *module-name***

The linker found a relocation from the near data section, which is placed in the root node, to the data section of an overlay node. This type of relocation is illegal.

- Error 613: MERGED Data relocation to static function is not resolvable by Overlay Manager.**
Reference at offset *hex-address* in *filename* , To Unit *filename*
- The overlay manager cannot resolve an indirect function call from the root node to a static function in a overlay node. To correct the problem, remove the **static** keyword from the function declaration.
- Error 614: More than one MERGED data section found**
- Only one near data section is allowed. This error occurs only when using **slink** to strip debug information from an executable generated by a third party linker. If you get this message, do not use **slink** to strip debug information from this executable.
- Error 615: Code hunk named `__MERGED`**
`slink` merges all hunks with the name `__MERGED` and performs relocations to this hunk relative to register A4. Do not name the code hunk `__MERGED` .
- Error 616: ALVs were generated**
- You have linked with the **noalvs** linker option, but the linker could not resolve all relocations without generating **ALV** instructions. Either the total code size is greater than 32K, or there are multiple code hunks. The executable will run, but the code section is not totally PC-relative.
- Error 617: MERGED data greater than 64K**
- If your program does not generate any additional messages, then the program will still run. However, a 16-bit data relocation record may not be able to reach its target, and if this happens, **slink** will generate an error.
- Error 618: Multiple OVERLAY usage—previous occurrences were ignored**
- You can only specify one overlay manager.
- Error 619: Ignoring null OVERLAY list**
- You did not specify any files in the overlay list.

Error 620: Missing '#' at end of OVERLAY list

Enter a pound sign (#) at the end of the overlay tree. For more information on creating overlay trees, see Chapter 8, "Compiling and Linking Your Program."

Error 621: Conflicting integer sizes found

Some modules were compiled using short integers and some were compiled using long integers. You cannot mix integer sizes in an executable. Alternatively, you may have linked in the wrong version of the library.

Error 622: Conflicting math types found

You may have compiled the various object modules with conflicting math options. All object modules must be compiled with the same math library. Alternatively, you may have linked in the wrong math library.

Error 623: Regargs function *function* called through overlay manager. Parameters passed in registers to this function will be destroyed.

The `regargs` function passes parameters in scratch registers, but the overlay manager does not preserve scratch registers. You cannot call a `regargs` function across an overlay node.

Error 624: Absolute reference to *symbol* module: file *filename*

If you reference far data in a module linked with `cres.o` or when you are generating a shared library, your program may function improperly unless the data are read-only. `slink` cannot determine if the reference is read-only, so it generates this warning.

Error 625: Proper math library has not been included

You have not linked in the math library that is needed for the current math options.

Error 626: Libcode used on module *module*

You have compiled the module with the `libcode` compiler option that was used on the module, but the module is being linked into an executable. Use the `libcode` option for generating shared libraries only.

Error 627: Near references found in executable that has a module compiled with FARONLY option

You cannot mix modules compiled with `data=faronly` and modules that refer to near data.

Enabling Suppressed Messages

Many messages are suppressed by default. The **strict** and **ansi** compiler options turn on some of these messages, but you must use the **warn** option to enable the remaining messages. The **ansi** option enables the following messages:

18 51 70 108 111 137 162 176 180 213

The **strict** option enables the following messages:

18 70 111 137 159 163 176 179 187 220
51 108 120 149 162 164 178 180 213

The following messages can be enabled only with the **warn** option:

22 148 156 165 190 195 212

The following table lists each message that is suppressed by default and indicates which compiler options enable the message.

Number	Message Text	ansi	strict	warn
18	sizeof operator used in preprocessor condition	X	X	X
22	structure used as function argument			X
51	C++ comment detected	X	X	X
70	unrecognized escape sequence	X	X	X
108	zero-length arrays are not an ansi feature	X	X	X
111	non-portable enum type specified	X	X	X
120	Integral type mismatch: possible portability problem Expecting "type1", found "type2"		X	X
137	ansi limits #line numbers to between 1 and 32767	X	X	X

(continued)

Number	Message Text	ansi	strict	warn
148	use of incomplete struct/union/enum tag "name" See line <i>number</i> file <i>filename</i>			X
149	incomplete struct/union/enum tag in prototype scope "name"		X	X
156	operation/comparison of pointer to "int" and pointer to "type"			X
159	use of unary minus on unsigned value		X	X
162	non-ansi use of ellipsis punctuator	X	X	X
163	initialization of auto struct, union, or array		X	X
164	& applied to array		X	X
165	use of narrow type in prototype			X
176	implicitly promoted formal "name" conflicts with prototype See line <i>number</i> file " <i>filename</i> "	X	X	X
178	indirect call without indirection operator		X	X
179	narrow type used in old-style definition		X	X
180	no space between macro name and its replacement list	X	X	X
187	negative value assigned to unsigned type		X	X
190	#include ignored because header already included See line <i>number</i> file " <i>filename</i> "			X
195	nested comment detected			X
212	item "name" already declared See line <i>number</i> file " <i>filename</i> "			X
213	empty argument to preprocessor macro	X	X	X
220	old-fashioned assignment operator		X	X

Appendix 3

Implementation-Defined Behavior

267	<i>Introduction</i>
268	<i>F.3.1 Translation</i>
268	<i>F.3.2 Environment</i>
268	<i>F.3.3 Identifiers</i>
269	<i>F.3.4 Characters</i>
269	<i>F.3.5 Integers</i>
270	<i>F.3.6 Floating-Point</i>
270	<i>F.3.7 Arrays and Pointers</i>
271	<i>F.3.8 Registers</i>
271	<i>F.3.9 Structures, Unions, Enumerations, and Bit-Fields</i>
272	<i>F.3.10 Qualifiers</i>
272	<i>F.3.11 Declarators</i>
272	<i>F.3.12 Statements</i>
272	<i>F.3.13 Preprocessing Directives</i>
273	<i>F.3.14 Library Functions</i>
278	<i>F.3.15 Locale-Specific Behavior</i>

Introduction

The *American National Standard for Information Systems — Programming Language — C* (ANSI Standard X3J11/90-013) allows an implementation to define its own behavior in certain areas. *Implementation-defined behavior* is defined by the ANSI Standard as “behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation. . .” (p.3). This appendix describes the SAS/C Compiler behavior for those areas listed in Section F.3, “Implementation-Defined Behavior,” of the ANSI Standard.

This appendix is organized like Section F.3.

F.3.1 Translation

- Diagnostics consist of a line printed to the standard output stream containing the following elements:

filename line class number: text

where:

filename is the name of the file that caused the diagnostic

line is the number of the line that caused the diagnostic

class is either **Error** or **Warning**

number is an integer identifying the diagnostic

text is the diagnostic text.

Some diagnostics contain additional information in a second line. If present, this second line is indented to the same position as the text portion of the first line.

F.3.2 Environment

- The compiler conforms to the ANSI Standard for a hosted environment as documented in Section 2.1.2.2 of the ANSI Standard, plus the extension of accepting a **WBStartup** structure as described under **main** in the *SAS/C Development System Library Reference, Version 6.0*.
- The compiler recognizes the streams **stdin**, **stdout**, and **stderr** as interactive devices.

F.3.3 Identifiers

- By default, the number of significant characters for identifiers without external linkage is 31. However, this number can be changed to any value up to 255 with the **idlen** compiler option.
- The number of significant characters for identifiers with external linkage is the same as the number for identifiers without external linkage.
- Case distinctions are significant in identifiers with and without external linkage.

F.3.4 Characters

- The source and execution character sets consist of the characters required by the ANSI Standard. In addition, national characters (characters with accents) are accepted as valid alphabetic characters unless the `ansi` compiler option is used. See *The AmigaDOS Manual, 3rd Edition* (Commodore-Amiga, Inc. 1991) for a complete definition of the Amiga character set.
- No shift states are used for encoding multibyte characters.
- Each character in the execution character set consists of 8 bits.
- The source character set maps 1-to-1 to the execution character set.
- All wide character constants map directly into the execution character set. Thus, all wide character constants have the same value as their corresponding character in the execution character set.
- The `C` locale is used as the current locale.
- A plain `char` item is by default `signed`, but this default can be changed to `unsigned` with the `unschar` compiler option.

F.3.5 Integers

- Integers are represented by two's-complement numbers. The range of possible values are specified in the header file `limits.h` and in Chapter 12, "How Does the Compiler Work?" The most-significant bit in the representation of signed integers is the sign bit and determines whether the number is positive or negative.
- When an integer is converted to a shorter signed integer, the high-order bytes are discarded. The bit pattern of the remaining bytes is unchanged. The bit pattern of an unsigned number is not changed when it is converted to a signed integer of equal length.
- The following list covers the results of bitwise operations on signed integers:

~ (bitwise NOT)

The bits are inverted; that is, 1 bits are set to 0, 0 bits are set to 1.

>> (shift right)

The bits are shifted to the right. The sign bit (uppermost bit) is used to fill the vacated bit positions on the left.

<< (shift left)

The bits are shifted to the left. The vacated bit positions on the right are filled with zeroes.

& (bitwise AND)

If the corresponding bits in both operands are 1, then the corresponding bit in the result is set to 1; otherwise, it is set to 0.

| (bitwise OR)

If either of the corresponding bits in the operands are 1, then the corresponding bit in the result is set to 1; otherwise, it is set to 0.

^ (bitwise XOR)

If the corresponding bits in the operands are not alike, then the corresponding bit in the result is set to 1; otherwise, it is set to 0.

- In integer division, the remainder is either 0 or has the same sign as the dividend.
- In a right shift of a negative-valued signed integral type, the sign bit is used to fill the vacated bit positions on the left. That is, the result retains the sign of the left operand.

F.3.6 Floating-Point

- Floating-point numbers are represented using the IEEE standard representation, unless the `math=ffp` option is on. In the latter case, the Motorola Fast Floating Point representation is used. The range of possible values for floating-point numbers is specified in the header file `float.h` and in Chapter 12, “How Does the Compiler Work?”
- If an integral number is converted to a floating-point number that cannot exactly represent the original value, the number is rounded for all `math` options except `math=coprocessor`. For `math=coprocessor`, the number is truncated. You can change the behavior for the `math=coprocessor` option by modifying the source code to the `__fpinit.c` function, located in `sc:source/_fpinit`. However, choosing another rounding mode affects the conversion from floating-point to integer such that the conversion does not adhere to the ANSI Standard. Specifically, the following statement might assign the integer value 6 to `i` if you specify `math=coprocessor`, and the coprocessor’s rounding mode is not set to truncate:

```
int i = (int)5.5;
```

- Conversions from `double` to `float` round or truncate in the same manner as conversions from integral types to floating-point types.

F.3.7 Arrays and Pointers

- The type of integer required to hold the maximum size of an array is `unsigned int`.
- When a pointer is converted to `integer`, the resulting bit pattern is as if an object of type `unsigned long` with the same bit pattern as the pointer had been converted to the integer type.

- When an integer is converted to a pointer, the resulting bit pattern is the same as if the integer had been converted to an **unsigned long**.

F.3.8 Registers

- Under normal circumstances, there can be up to 4 integer register variables, 3 pointer register variables, and 5 floating point register variables. Using the **cover** compiler option, the **stackext** compiler option, the **__stackext** keyword, or **__aligned** keyword on a function definition costs one pointer register variable. Using the **data=faronly** compiler option adds an additional pointer register variable. The **register** keyword is ignored when the global optimizer is used because registers are allocated to variables by the global optimization phase. If you specify the **autoreg** option, the code generator selects additional register variables for you if registers are available after assigning your choices to registers.

F.3.9 Structures, Unions, Enumerations, and Bit-Fields

- If a member of a union is accessed using a member of a different type, the result is undefined. If optimization is used, the compiler may attempt to keep the value stored in one member of a union in a register, and a subsequent access to a different member may get an old value.
- Structure members of type **char** can appear at any byte offset. Structure members of other simple types must be aligned on an even byte boundary, and padding is inserted if necessary. Any structure or union member declared with the **__aligned** keyword is aligned on a four-byte boundary, relative to the start of the structure or union, with padding inserted as necessary. Substructures and subunions have the same alignment requirements as their most restrictive members. Structures and unions are padded at the end to the boundary of their most restrictive member. For more information on structure padding, see Chapter 13, “Writing Portable Code.”
- **int** bitfields are treated as **unsigned int** bitfields.
- The order of allocation of bitfields for an **int** is from left to right. Bitfields never cross storage unit boundaries.
- The values of an enumeration type are normally of type **int**. However, the SAS/C Compiler allows the declaration of **char enum**, **short enum**, and **long enum** types. Enumeration types so declared are of the specified type. For more information on using enumerated types,

see Chapter 11, “Using Amiga Specific Features of the SAS/C Language.”

F.3.10 Qualifiers

- Any reference to an object whose type is qualified by the keyword `volatile` is considered an access to that object.

F.3.11 Declarators

- There is no limit on the number of declarators that can modify an arithmetic, structure, or union type.

F.3.12 Statements

- There is no limit to the number of `case` values in a `switch` statement.

F.3.13 Preprocessing Directives

- The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Such a character can have a negative value unless the `unschar` compiler option is active.
- The compiler defines a search path for `#include` files that are included with angle brackets (`<>`) around their names. This path is defined as follows:
 1. directories specified with the `includedirectory` compiler option
 2. the logical assignment `INCLUDE:.`
- The compiler uses a slightly different search path for `#include` files that are included with double quotes (`""`) around their names. This path is defined as follows:
 1. the current directory
 2. the directory of the file containing the `#include` statement
 3. directories specified with the `includedirectory` compiler option
 4. the logical assignment `INCLUDE:.`
- Source file character sequences are not mapped. The string between the angle brackets (`<>`) or double quotes (`""`) is passed to the file system.

- The behavior of the `libcall`, `syscall`, and `tagcall` `#pragma` directives is described in the *SAS/C Development System Library Reference*. The behavior of the `msg` and `flibcall` `#pragma` directives is described in Chapter 11, “Using Amiga Specific Features of the SAS/C Language.”
- The definitions for the `__DATE__` and `__TIME__` preprocessor macros contain the date and time as specified by the `DateStamp` AmigaDOS function. This function always returns a value, so the date and time are always available.

F.3.14 Library Functions

- The `NULL` pointer constant to which the macro `NULL` expands is `0L`.
- The diagnostic printed by the `assert` function is of the form:

`Assertion (condition) failed in file filename at line number`

where:

`condition` is the condition passed to the `assert` macro

`filename` is the current value of `__FILE__`

`number` is the current value of `__LINE__`

The `assert` function calls `abort` after printing the above line. For more information, refer to the description of the `assert` function in the *SAS/C Development System Library Reference*.

- The sets of characters tested for by the `isalnum`, `isalpha`, `isctrl`, `islower`, `isprint`, and `isupper` functions is defined by the entries in the external array `__ctype`. The following table lists the hexadecimal values for each of the characters tested for by these functions.

Function Name	Character Type	Hexadecimal Values
<code>isalnum</code>	alphabetic or numeric	<code>0x30</code> to <code>0x39</code> , <code>0x41</code> to <code>0x5a</code> , <code>0x61</code> to <code>0x7a</code>
<code>isalpha</code>	alphabetic	<code>0x41</code> to <code>0x5a</code> , <code>0x61</code> to <code>0x7a</code>
<code>isctrl</code>	control	<code>0x00</code> to <code>0x1f</code> , <code>0x7f</code>

(continued)

Function Name	Character Type	Hexadecimal Values
islower	lowercase	0x61 to 0x7a
isprint	printable	0x20 to 0x7e
isupper	uppercase	0x41 to 0x5a

- The mathematics functions return zero on domain errors.
- On underflow range errors, the mathematics functions set the integer expression **errno** to the macro **ERANGE**.
- Zero is returned when the **fmod** function has a second argument of zero.
- The set of signals for the **signal** function are the minimum set defined by the ANSI Standard: **SIGABRT**, **SIGFPE**, **SIGILL**, **SIGINT**, **SIGSEGV**, and **SIGTERM**.
- Refer to the description of the **signal** function in the *SAS/C Development System Library Reference* for the semantics of the signals.
- By default, the **SIGINT** signal results in calling the **_CXBRK** function, which terminates the program. Also by default, the **SIGFPE** signal results in calling the **_CXFPE** function, which sets **errno** and returns to the caller causing the exception. By default, all other signals are ignored.
- The default handling is reset if the **SIGILL** signal is received by a handler specified to the **signal** function.
- The last line of a text stream does not require a terminating newline character. No newline character is generated if one is not explicitly written.
- Space characters that are written out to a text stream immediately before a newline character appear when read in.
- The implementation does not append any **NULL** characters to data written to a binary stream.
- The file position indicator of an append mode stream is initially positioned at the beginning of a file.
- A write on a text stream does not cause the associated file to be truncated beyond that point (see Section 4.9.3 of the ANSI Standard). However, some of the modes truncate the file upon open. The following table lists whether a file is truncated after a write.

Mode	File Truncated?	Comments
"r" or "ra"	N/A	No writes allowed.
"w" or "wa"	No	The file is truncated at open time only.
"a" or "aa"	No	All data are written after EOF.
"r+" or "ra+"	No	
"w+" or "wa+"	No	
"a+" or "aa+"	No	All data are written after EOF.
"rb"	N/A	No writes allowed.
"wb"	No	The file is truncated at open time only.
"ab"	No	All data are written after EOF.
"rb+"	No	
"wb+"	No	
"ab+"	No	All data are written after EOF.

- A call to the **fopen** function opens a buffered I/O stream. For writes, data are stored into the buffer until it is full, and then the data are written out. A call to the **fflush** function forces a write of any buffered data.

You can use the library routines **setbuf** and **setvbuf** to change between buffered, unbuffered, and line buffered modes. For more information, refer to the description of these functions in the *SAS/C Development System Library Reference*.

- A zero-length file can be created by either creating a file and not writing anything to it or by truncating an existing file.
- In general, a file can be opened multiple times as long as a previous call to open the file did not physically create the file. However, some devices or file-handlers may prevent multiple opens. Refer to the documentation for the device or file-handler that you are using.
- A file cannot be deleted if there are any outstanding locks or file handles on the file. In this case, the **remove** function sets **errno** and **_OSERR** and returns a nonzero value.
- The output for **%p** conversion specification for the **fprintf** function is the pointer's value in hexadecimal.

- The input for the **%p** conversion specification for the **fscanf** function is a hexadecimal pointer, optionally preceded by **0x** or **0X**.
- A minus (–) character that is neither the first nor the last character in the scanlist for a **%[** conversion specification in the **fscanf** function is treated as a subrange specifier. That is, **%[a–d]** is equivalent to **%[abcd]**.
- The functions **fgetpos** and **ftell** set the variable **errno** to the value of **EBADF** for a bad file handle. Otherwise, **errno** is set as described in the *SAS/C Development System Library Reference* for the function **lseek**.
- The **perror** function generates the following messages:

errno Value	Description
0	Unknown error code
EPERM	User is not owner
ENOENT	No such file or directory
ESRCH	No such process
EINTR	Interrupted system call
EIO	I/O error
ENXIO	No such device or address
E2BIG	Arg list is too long
ENOEXEC	Exec format error
EBADF	Bad file number
ECHILD	No child process
EAGAIN	No more processes allowed
ENOMEM	No memory available
EACCES	Access denied

(continued)

errno Value	Description
EFAULT	Bad address
ENOTBLK	Bulk device required
EBUSY	Resource is busy
EEXIST	File already exists
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid argument
ENFILE	No more files (units) allowed
EMFILE	No more files (units) allowed for this process
ENOTTY	Not a terminal
ETXTBSY	Text file is busy
EFBIG	File is too large
ENOSPC	No space left
ESPIPE	Seek issued to pipe
EROFS	Read-only file system
EMLINK	Too many links
EPIPE	Broken pipe
EDOM	Math function argument error
ERANGE	Math function result is out of range

- If the functions `calloc`, `malloc`, or `realloc` are passed a size of zero, the allocation fails, and the function returns a `NULL` pointer. **errno** is set to the value of **EINVAL**.
- The `abort` function closes all open and temporary level 2 I/O files.
- The value passed to the `exit` function is the value that is passed back to the system. For **EXIT_SUCCESS**, the value returned is 0. For **EXIT_FAILURE**, the value returned is 20.

- The list of environment names are in the system file-handler **ENV:**. Environment name values are modified through the **putenv** function.
- The contents and mode of execution of the string passed to the AmigaDOS system by the **system** function is as follows:
 - Under AmigaDOS Version 1.3, the AmigaDOS function **Execute** is called.
 - Under AmigaDOS Version 2.0 or higher, the AmigaDOS function **System** is called.

The **system** function returns the value returned from the AmigaDOS **Execute** or **System** function. If the command processor cannot be invoked, **system** returns a -1.

- The **strerror** function returns the same error messages as the **perror** function. The messages returned by the **perror** function are listed above.
- The default time zone is **CST6**, and the default for Daylight Savings Time is 0.
- The era for the **clock** function is based on the first call to the **clock** function. The first call returns a value of zero, and subsequent calls measure the processor time from that starting point. For more information, refer to the description of the **clock** function in the *SAS/C Development System Library Reference*.

F.3.15 Locale-Specific Behavior

- The execution character set contains all ASCII characters in all locales.
- The direction of printing is from left to right.
- The period (.) is the decimal-point character.
- Refer to the description of the **is.....** functions in the *SAS/C Development System Library Reference* for information on the implementation-defined aspects of character-testing and case-mapping functions.
- The collation sequence for ASCII characters is used as the collation sequence of the execution character set.
- The SAS/C Development System uses the normal U.S. time and date conventions for the **C** locale.

Type	Values
Weekday name	Sun Mon Tue Wed Thu Fri Sat
Month name	Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
Date/Time format	DDD MMM dd hh:mm:ss YYYY Tue Jun 16 14:12:22 1992

Appendix 4

Converting From Aztec C Options to SAS/C® Options

Table A4.1 lists each of the Aztec 5.0 options and their equivalent SAS/C options (if available). Some Aztec C options do not have equivalent SAS/C options. Also, for some SAS/C options listed in this table, you must specify additional options before the ones listed in this table will take effect. For example, you must specify the **genprotos** option before the **genprotoparms** option will take effect.

Table A4.1 *Converting From Aztec C Options to SAS/C Options*

Aztec C Option	SAS/C Option	Comments
-3	no equivalent	
-5	no equivalent	
-a	disasm=filename noobjname	These options generate assembly output.
-at	disasm=filename debug=line noobjname	The debug option adds C source to the assembly output.
-bd	stackcheck	stackcheck is the default setting. You can turn off stack checking with nostackcheck .
-bs	debug=level	You can specify one of five levels of debug information.
-c2	cpu=68020	
-d	define symbol=value	
-fa	math=ieee	You must also link with the appropriate math library.
-ff	math=ffp	You must also link with the appropriate math library.
-fm	math=standard	You must also link with the appropriate math library.

(continued)

Table A4.1 (continued)

Aztec C Option	SAS/C Option	Comments
-f8	math=68881	You must also link with the appropriate math library.
-hi	gst=gst-filename	
-ho	makegst=gst-filename	
-i	includedirectory=filename	
-k	no equivalent	
-ma		This option is on by default.
-mb	no equivalent	
-mc	code=far	
-md	data=far	
-me	no equivalent	
-mm	stringmerge	stringmerge forces the compiler to have only one copy of identical strings and places these strings in the code section. stringmerge also places data declared static const into the code section.
-ms	nostringmerge	
-o	objectname=filename	
-pa	ansi trigraph	
-pb		Bitfields are unsigned by default. Use the signed keyword to declare a signed bitfield.
-pc	ignore=132	This option suppresses the warning for tokens after #endif.
-pe	no equivalent	
-pl	noshortint	
-po	oldpp	
-pp	unsignedchar	

(continued)

Table A4.1 (continued)

Aztec C Option	SAS/C Option	Comments
-ps	shortintegers	
-pt	trigraph	
-pu	no equivalent	
-qa	genprotoparms	
-qf	errorrexx	
-qp	nogenprotostatics	
-qq	noverbose noversion	
-qs	nogenprotoexterns	
-qv	no equivalent	
-r4	data=faronly	This option uses register A4 for register variables. Do not use the __near or __chip keywords if you use this option.
-sa		This option is always on.
-sb	no equivalent	Include string.h in your source code.
-sf	optimize	
-sm	no equivalent	Include the appropriate header file in your source code.
-sn		This option is always on.
-so	optimize	
-sp		The cleanup overhead reduction enhancement is always on unless you specify debug=sf or debug=ff .
-sr	optimize	
-ss	stringmerge	
-su		This option is the default setting. You can disable this option with the noautoreg option.

(continued)

Table A4.1 (continued)

Aztec C Option	SAS/C Option	Comments
-wa		This option is the default setting. You can disable this option with the ignore=88 option.
-wd		
-we	error=all	This option turns all warnings into errors. To make a specific message <i>n</i> an error message, specify error=n .
-wl		This option is the default setting.
-wn	ignore=225	
-wo		This option is the default setting.
-wp		This option is the default setting. You can disable this option with the ignore=100 option.
-wq	no equivalent	Use command line redirection.
-wr		This option is the default setting. You can disable this option with the nowarnvoidreturn or ignore=85 option.
-ws	ignore=all	This option turns off all warnings.
-wu		This option is the default setting. You can disable this option with the ignore=93 option.
-ww	maxerr=n	The number <i>n</i> specifies the new error limit.

Appendix 5

Converting from Version 5 to Version 6

285	<i>Introduction</i>
285	<i>Upgrading from Previous Versions</i>
286	<i>Converting Compiler Options</i>
294	<i>Specifying Version 6 Libraries</i>

Introduction

This appendix lists the Version 6 options and libraries that are equivalent to the Version 5 options and libraries.

Upgrading from Previous Versions

You should install Version 6 in a separate location from previous versions of the compiler. Modify your startup sequence files to either remove the **LC:** directory from your path or ensure that the directory **SC:C** is on your search path before **LC:**.

In addition, make sure that **include:** and **lib:** get assigned to the correct include and link library directories for Version 6. The compiler for Version 6 does not use the **QUAD:** device used by previous versions.

In general, it is a good idea to recompile all your code when you convert to Version 6. However, because the Version 6 libraries are ANSI-compliant, your old programs may not compile or may generate many warning messages. Appendix 1, "Solving Common Problems," provides solutions to some of the problems you may encounter in converting to Version 6.

You may want to use the **sc5** command to convert older, seldom-used projects. The **sc5** command accepts Version 5 options and reads default options from the file **sasopts**. However, it is recommended that you become familiar with and begin using Version 6 as soon as possible.

You can use the **lctosc** conversion utility to convert your Version 5 **sasopts** file to a Version 6 **scoptions** file and to determine quickly the Version 6 equivalent of a Version 5 option. The **lctosc** utility reads the **sasopts** file in the current directory, reads any command-line

options, converts the options to Version 6 options, and writes the results to standard output. For example, to convert the **sasopts** file in the current directory to a Version 6 **scoptions** file, enter the following command at the Shell prompt:

```
lctosc >scoptions
```

To determine the Version 6 equivalent of the Version 5 option **-cs**, enter the following command:

```
lctosc -cs
```

For code compiled with previous versions of the compiler, the new version of CodeProbe can read the line number information but cannot read the symbol information from the executables. Recompile your code with the Version 6 compiler to produce usable debugging information for symbols.

If you get unresolved symbols when you link your application, you may be referring to symbols whose name has changed so as to make the compiler conform to the ANSI Standard. In this case, look up the symbol name in the *SAS/C Development System Library Reference, Version 6.0* to determine the new name, and change any references to the symbol name in your code to the new name. Alternatively, you can choose to use the **define** linker option to force all references to the old name to refer instead to the new name.

Converting Compiler Options

Table A5.1 lists each of the Version 5 options and the equivalent Version 6 option (if available). Some Version 5 options do not have equivalent Version 6 options. Also, for some Version 6 options listed in this table, you must specify additional options before the ones listed in this table will take effect. For example, you must specify the **link** option before any of the **math=type** options will take effect, and you must specify the **genprotos** option before the **genprotoparms** option will take effect.

Table A5.1 Converting Version 5 Options to Version 6 Options

Version 5 Option	Version 6 Option	Description
-+	verbose	Display commands as executed
-.	noversion	Suppress execution message
-ab	bss=chip	Force all uninitialized data (BSS) to chip memory
-ac	code=chip	Force all code to chip memory
-ad	data=chip	Force all initialized data to chip memory
-b	data=near	Base relative data (16-bit offsets)
-b0	data=far	Non-base relative data (32-bit offsets)
-b1	data=near	Base relative data (same as -b)
-ba	data=auto	Automatic non-base relative on data overflow
-c+	idlen=100	Allow longer identifiers for C++
-ca	ansi	Maximize ANSI compatibility
-cc	commentnest	Allow nested comments
-cd	ignore=77	Allow \$ in identifiers
-ce	noerrorsources	Suppress source line printing for errors
-cf		Require function prototypes ¹
-ci	nomultipleincludes	Suppress multiple includes of same file
-ck	ansi	Suppress non-ANSI keywords (chip , far , near)
-cl	Use __aligned keyword ²	Align all externs on longword boundary
-cm	multiplecharconstants	Allow multiple character constants
-co	oldpp	Enable old style preprocessor
-cp		Allow #if to span files
-cq	structequivalence	Allow passing of equivalent structure silently

(continued)

¹**sc5** did not require function prototypes. By default, Version 6 requires function prototypes. To suppress the Version 6 messages dealing with prototypes, specify **ignore=100+161+154**.

²Use the **__aligned** keyword to force individual objects to longword boundaries.

Table A5.1 (continued)

Version 5 Option	Version 6 Option	Description
-cr		Disallow <code>__asm</code> keyword on function definitions
-cs	stringmerge	Create only one copy of identical strings
-ct		Obsolete - does nothing
-cu	unsignedchar	Force all char declarations to unsigned char
-cw	nowarnvoidreturn	Shut off warning for return without a value
-cx	noexternaldefs	Treat all global declarations as externs
-C	onerror=continue	Continue on error
-d	debug=line	Enable debugging
-d0	nodebug	Disable debugging
-d1	debug=line	Enable debugging - dump line table
-d2	debug=symbol	Generate symbol information
-d3	debug=symbolflush	Generate symbol information and flush registers at every line
-d4	debug=full	Generate full symbol information
-d5	debug=fullflush	Generate full symbol information and flush registers at line
-dsym=val	define sym=val	Define preprocessor symbol to a value
-E	errorrexx	Invoke Editor on error
-Efilename		Invoke Editor on error, <i>filename</i> is an errorfile name
-e		Enable Asian character set (default is Japanese)
-e0		Enable Japanese character set (same as -e)
-e1		Enable Chinese character set
-e2		Enable Korean character set
-f	math=ffp	Use Motorola FFP (forces -fs)
-f8	math=68881	Generate code for Motorola 68881

(continued)

Table A5.1 (continued)

Version 5 Option	Version 6 Option	Description
-fd	precision=double	Use double precision for both float and double
-ff	math=ffp	Use Motorola FFP
-fi	math=ieee	Generate code for IEEE libraries
-fl	math=standard	Use standard SAS/C floating point library
-fm	precision=mixed	Use single precision for float, double for double
-fs		Use single precision for both float and double
-gc	xrefsystem	Cross-reference compiler-provided files
-gd	xrefmacros	Cross-reference defined symbols
-ge		List excluded lines
-gh	listheaders	List header files
-gi	listsystem	List included files
-gm	listmacros	List macro expansions
-gn	listnarrow	Print narrow lines
-go	errorlisting errorconsole	Put errors and warnings to both stderr and listing file
-gs	list	List source
-gx	xreference	Produce cross reference listing
-g=filename	listfile=filename	Generate listing file with name <i>filename</i>
-Hname	gst=gst-filename	Read in precompiled header file <i>name</i>
-hb	bss=fast	Force all uninitialized data (BSS) to fast RAM
-hc	code=fast	Force all code to fast RAM
-hd	data=fast	Force all initialized data to fast RAM
-ix	incdirectory=filename	Specify directory to search for includes
-jn	ignore=n	Disable message number <i>n</i>

(continued)

Table A5.1 (continued)

Version 5 Option	Version 6 Option	Description
-jne	error=<i>n</i>	Make message number <i>n</i> an error instead of a warning
-j*e	error=all	Turn all warnings into errors
-jni	ignore=<i>n</i>	Disable message number <i>n</i> (same as -jn)
-j*i	ignore=all	Disable all warnings
-jnw	warn=<i>n</i>	Enable message number <i>n</i> as a warning
-ja	noerrorhighlight	Suppress ANSI escape codes from error line output
-L+	object=object-file library=link-library	Specify additional linker objects
-La	addsym	Add symbol information when linking
-Lc	smallcode	Use small code memory model (16-bit jumps)
-Ld	smalldata	Use small data memory model (16-bit data offsets)
-Lf	math=ffp	Link with FFP math library
-Lh	maphunk	Produce linker hunk map
-Li	stdio	Link with <code>_tinymain</code> instead of <code>_main</code>
-Ll	maplib	Produce linker library map
-Lm	math=math-type	Link with appropriate math library
-Ln	stripdebug	Suppress debugging information in executables
-Lo	mapoverlay	Produce linker overlay map
-Ls	mapsym	Produce linker symbol map
-Lt	smallcode smalldata verbose	Use typical link options (same as -Lc , -Ld , -Lv , or -Lcdv)
-Lv	verbose	Print verbose linking information
-Lx	mapxref	Produce linker cross-reference

(continued)

Table A5.1 (continued)

Version 5 Option	Version 6 Option	Description
-l	Use <code>__aligned</code> keyword. ¹	Align objects on longword boundaries
-M	<code>modified</code>	Compile only modified source files
-m	<code>cpu=any</code>	Generate code for Motorola 68000
-m0	<code>cpu=68000</code>	Generate code for Motorola 68000 (same as -m)
-m1	<code>cpu=68010</code>	Generate code for Motorola 68010
-m2	<code>cpu=68020</code>	Generate code for Motorola 68020
-m3	<code>cpu=68030</code>	Generate code for Motorola 68030
-ma	<code>cpu=any</code>	Generate code for all Motorola processors (same as -m)
-mc		Disable cleanup overhead reduction enhancement
-ml	<code>libcode</code>	Generate code for resident library use
-mp	<code>absfuncpointer</code>	Generate 32-bit references to functions
-mr	<code>noautoregister</code>	Disable automatic registerization
-ms	<code>optsize</code> <code>optimize</code>	Generate code optimized for space
-mt	<code>opttime</code> <code>optimize</code>	Generate code optimized for time
-N		Disable linking from editor
-n	<code>identifierlength=8</code>	Retain only 8 characters for identifiers
-nn	<code>identifierlength=n</code>	Specify maximum length for identifiers
-O	<code>optimize</code>	Invoke global optimizer
-ox	<code>objectname=filename</code>	Place object file in <i>filename</i>
-Px	<code>programname=filename</code>	Use <i>filename</i> as the name of the final executable module

(continued)

¹Use the `__aligned` keyword to force individual objects to longword boundaries.

Table A5.1 (continued)

Version 5 Option	Version 6 Option	Description
-p	preprocessonly	Preprocess only
-pe	nogenprotostatics	Generate prototypes for external functions
-ph	makegst=filename	Generate precompiled header file
-pi		Include identifier names in generated prototypes
-pp	genprotoparms	Generate prototypes with __PARMS for portability
-pr	genproto	Generate prototypes for external and static functions
-ps	nogenprotoexterns nogenprotodataitems	Generate prototypes for static functions only
-pt	nogenprototypedefs	Suppress use of typedefs when generating prototypes
-qx		Place quad file in location x
-qne	maxerr=n	Quit compilation after n error/warnings
-qnw	maxwarn=n	Quit compilation after n warnings
-q	maximumwarn=1 maximumerrors=1	Quit after 1 error or 1 warning (same as -q1e1w)
-r	code=near	Default function calls to near (PC-relative)
-r0	code=far	Default function calls to far (Absolute)
-r1	code=near	Default function calls to near (Same as -r)
-rr	parms=register	Pass and receive parameters in registers
-rs	parms=stack	Pass and receive parameters on the stack
-rb	parms=both	Generate code for both register and stack conventions
-Rfilename	objectlib=filename	Place compiled objects into library filename

(continued)

Table A5.1 (continued)

Version 5 Option	Version 6 Option	Description
-s		Use default segment names text , data , and udata
-sb=name	bssname=name	Specify a name for uninitialized data (BSS) segment
-sc=name	codename=name	Specify a name for code segment
-sd=name	dataname=name	Specify a name for data segment
-t	startup=cback	Link with cback.o instead of c.o
-tb	startup=cback	Link with cback.o instead of c.o (same as -t)
-tc	startup=catch	Link with catch.o instead of c.o
-tcr	startup=catchres	Link with catchres.o instead of c.o
-tr	startup=cres	Link with cres.o instead of c.o
-t=x	startup=filename	Link with <i>filename</i> instead of c.o
-u		Undefine all preprocessor symbols
-v	nostackcheck	Disable stack checking code
-w	shortintegers	Default to short integers
-x	noexternaldefs	Treat all global declarations as externals (same as -cx)
-y	saveds	Load A4 with base address at start
-znnn	ppbuf=nnn	Set preprocessor expansion buffer size to <i>nnn</i>

Specifying Version 6 Libraries

Table A5.2 lists each of the Version 5 libraries and the equivalent Version 6 library. In Version 6, the standard libraries provide support for registerized parameters. To use registerized parameters, you must compile with the `parms=registers` option, but you do not need a special library.

Table A5.2
Specifying Version 6
Libraries

Version 5 Library	Version 6 Library
lc.lib	sc.lib
lcm.lib	scm.lib
lcmieee.lib	scmieee.lib
lcmffp.lib	scmffp.lib
lcmr.lib	scm.lib
lcms.lib	scms.lib
lcmsr.lib	scms.lib
lcm881.lib	scm881.lib
lcmr881.lib	scm881.lib
lcnb.lib	scnb.lib
lcr.lib	sc.lib
lcs.lib	scs.lib
lcsnb.lib	scsnb.lib
lcsr.lib	scs.lib

Index

A

- .a filename extension 78
- abbreviated menus 50
- abort function 277
- absfuncpointer compiler option 85, 177
- accelerator keys 50
- addressing data 176–177
 - chip data 176
 - modifying access to program code and data 176–177
 - reentrant and non-reentrant programs 176
- addsym linker option 121
- addsymbols compiler option 85
- _aligned keyword 155–156
 - affecting definitions of functions 155
 - automatic variable problems 156
 - ensuring stack alignment 156
 - example 156
 - pointer register variables 161
- Alt keys
 - equivalence 60
 - independent operation 60
- Amiga keys
 - caution against redefining 60
- Amiga-specific features of SAS/C language 153–169
 - C++ style comments 168
 - comma (,) operator 168
 - common model external data 168–169
 - enumerated types 167
 - equivalent structures 164–166
 - implicit structure references 164
 - managing stack space 160–161
 - national characters in variable names 168
 - nested comments 168
 - #pragma statements 161–163
 - sizeof operator 168
 - special keywords 154–159
 - unnamed unions 163–164
 - zero-length arrays 163–164
 - _aligned keyword 155–156
 - _asm keyword 157–158
 - _chip keyword 156–157
 - _far keyword 156–157
 - _inline keyword 159
 - _interrupt keyword 157
 - _near keyword 156–157
 - _regargs keyword 157–158
 - _savesd keyword 158–159
 - _stackext keyword 160–161
 - _stdargs keyword 157–158
- amigaguide.se, AREXX macro 69
- AmigaGuide, invoking 28
- ANSI compiler option 85, 181
- ANSI Standard
 - See also implementation-defined behavior
 - See also portable code, writing
 - default argument promotion rules 187
 - format for incoming command line 140
- ANSI-compliant C compilers 182
- AREXX interface 65–73
 - AREXX port 68
 - command summary 70–73
 - commands without keystroke equivalents 70–73
 - communicating with external programs 68–69
 - definition 73
 - executing a series of editor commands 65–68
 - keystroke equivalents for commands 70–73
- AREXX macros
 - amigaguide.se 69
 - assigning to keys 66
 - definition 73
 - deletetabs.se 69
 - examples 66–68, 69
 - findsym.se 69
 - indent.se 70
 - newline character (\n) 65
 - provided with the compiler 69–70

AREXX macros (*continued*)

- .se suffix required 65
- tolower.se 70
- toupper.se 70
- unindent.se 70

argc and argv

- ANSI Standard requirement 140
- Shell environment 135–136
- Workbench environment 136

arguments

- raising argument limits 140–141

argumentsize= compiler option 85–86

- See also memorysize= compiler option

arrays

- SAS/C Compiler behavior 270–271

asctime function 200–201

.asm filename extension 78

— _asm keyword

- affecting caller and callee 155
- Amiga-specific features 157–158

Asm modes 61–62

assembler= compiler option 78, 86

assert function 273

assign statements, adding to startup file 10

asterisk (*)

- Asm mode 62
- indicating depth of overlay tree 129
- indirection operator 155

at sign (@)

- preceding linker symbols 122, 158
- /AUTO keyword 138

Auto-Indent Mode, Options pull-down menu 62–63

autoinitialization function 145–147

- definition 145
- example 146
- level 1 or 2 I/O functions not used 147
- library bases automatically initialized 148–149
- order in which modules called 147
- _STI preceding function name 145
- _ctors array 146, 147

automatic storage class 180

automatic variables

- stack alignment problems 156

— _autoopenfail function

- calling when autoinitialized libraries cannot open 150–151
- source code 151

autoregister compiler option 86, 180, 271

autotermination function 145–147

- definition 145
- example 146
- level 1 or 2 I/O functions not used 147
- order in which modules called 147
- _STD preceding function name 145
- _dtors array 146, 147

Autowrap option 61–62

Aztec C options, converting 281–284

B

— _BackGroundIO external definition 143

backslash (\)

- preceding comment character 62

— _Backstdout external definition

- AmigaDOS functions requirements 143
- definition 143
- NULL file handle 143
- printing messages in the Shell 143

backup copies of installation diskettes 3

Backup File Mode option, Options pull-down menu 63

backward searching 42

batch compiler option 86

batch linker option 121

Beginning option

- Block pull-down menu 53

Beginning option, Block pull-down menu 51

bitfields

- SAS/C Compiler behavior 271

bitwise operations 269–270

BIX (Byte Information Exchange) 31

block operations 43–44, 53

- marking blocks 43
- unmarking blocks 44

Block pull-down menu

- Beginning 51, 53
- Copy 53
- Delete 52, 53
- End 51, 53
- Move 53
- Print 53
- Read 53
- Write 53

braces ({ })

- Auto-Indent Mode 62–63

bssmemory= compiler option 86, 177

bssname= compiler option 87
 bufsize linker option 121
 Build icon
 compiling and linking programs 18–19
 definition 14

C

c directory 6
 .c filename extension 78
 c.o startup module
 default module 142
 definition 140
 C++ style comments 168
 calloc function 141, 277
 Case Sensitive Characters option, Options
 pull-down menu 63–64
 case sensitive searches 42
 casting results of short integers 179
 catch.o startup module
 creating snapshot files 142
 definition 140
 catchres.o startup module
 debugging residentable programs 144
 definition 140
 saves compiler option and __saves
 keyword not allowed 159
 snapshot files 142, 144
 cback.o startup module
 char * __procname external definition
 142
 creating detachable programs 142–143
 definition 140
 extern BPTR __Backstdout external
 definition 143
 long __BackGroundIO external
 definition 143
 long __priority external definition 142
 long __stack external definition 142
 uses 142
 char * __procname external definition 142
 character constants 104
 character strings, replacing
 See replacing character strings
 character strings, searching
 See searching for character strings
 characters
 locale-specific behavior 278–279
 SAS/C Compiler behavior 268

chip data
 reentrant and non-reentrant programs
 176
 chip hunk 172
 __chip keyword 156–157
 declaring data items 176
 overriding data=auto compiler option
 90
 purpose and use 156–157
 chip linker option 121
 chip memory 157
 clock function 278
 /CLOSE keyword 138
 Close Window option
 Project pull-down menu 47
 Windows pull-down menu 54
 code= compiler option 87, 177
 code hunk 172
 codememory= compiler option 87, 177
 codename= compiler option 87
 CodeProbe
 See debugging programs
 colon (:)
 Asm mode 62
 colors, changing 54
 Column Display option, Options pull-down
 menu 63
 comma (,)
 separating filenames for sc command 78
 comma (,) operator in preprocessor
 directives 168
 commentnest compiler option 87–88, 168
 comments
 C++ style comments 168
 comment characters, Asm mode 62
 comment start sequence (/*) 168
 nested comments 168
 nested comments not allowed by ANSI
 168
 slashes (//) defining comment blocks 168
 common compiler option 88, 169
 common model external data 168–169
 compiler options 79–119
 See also compiler options, setting
 abbreviations 80
 absfuncpointer 85, 177
 addsymbols 85
 ANSI 85, 181
 argumentsize= 85–86
 assembler= 78, 86

compiler options (*continued*)

- autoregister 86, 180, 271
- batch 86
- bssmemory= 86, 177
- bssname= 87
- case-insensitivity 80
- code= 87, 177
- codememory= 87, 177
- codename= 87
- commentnest 87–88, 168
- common 88, 169
- constlibbase 88
- converting from Aztec C options
 - 281–284
- converting Version 5 options to Version 6
 - 286–293
- coverage 88–89, 271
- CPU= 89
- csource= 78
- data= 89–90, 132, 157, 177
- data=auto 90
- data=far 89, 182
- data=faronly 90, 161, 182
- datamem=chip 172
- datamemory= 90, 177
- dataname= 91
- data=near 89, 172
- debug= 91, 173
- debug=full 91
- debug=fullflush 91
- debug=line 91
- debug=symbol 91
- debug=symbolflush 16, 17, 91
- define= 92
- disassemble= 92
- error= 92
- errorconsole 92
- errorhighlight 93
- errorlist 93
- errorrex 16, 17, 93
- errorsources 93
- externaldefs 93
- findsymbol= 93
- from 94
- genprotodataitems 94
- genprotoexterns 94
- genprotofile= 94
- genprotoparameters 94
- genprotos 94–95
- genprotostatistics 95
- genprototypedefs 95
- globalsymboltable= 96
- GSTimmediate 96
- icons 96–97
- identifierlength= 97
- ignore= 97
- includedirectory= 97
- keepline 97
- keyword options 79
- libcode 98, 159
- library= 78, 98
- link 16, 17, 24, 98
- linkeroptions= 98–99, 120
- list 99
- listfile= 99
- listheaders 99
- listinclude 99–100
- listmacros 100
- listnarrow 100
- listsystem 100
- makeglobalsymboltable= 100
- map 100
- mapfile= 100
- mapjunk 101
- maplib 101
- mapoverlay 101
- mapsymbols 101
- mapxreference 101
- math= 102
- math=coprocessor 270
- math=ffp 270
- math=ieee 270
- math=standard 16, 17
- maximumerrors= 102
- maximumwarnings= 102
- memorysize= 103
- modified 104
- multiplecharacterconstants 104–105
- multipleincludes 105
- noautoreg 180
- object= 78, 105
- objectlibrary= 106
- objectname= 106
- oldpreprocessor 106–107
- onerror= 107–108
- optimize 108
- optimizealias 108
- optimizercomplexity= 108
- optimizerdepth= 109
- optimizerglobal 109

- optimizerinline 109
- optimizerinlocal 109
- optimizerloop 109
- optimizerpeephole 109
- optimizerrecurdepth= 110
- optimizersize 110
- optimizertime 110
- overriding settings in scoptions file 79
- parameters= 110–111
- parameters=both 202
- parameters=register 202
- precision= 111
- preprocessor symbols defined 120
- preprocessorbuffer= 111
- preprocessoronly 111–112
- programname= 112
- resetoptions 79, 112
- SAS/C Compiler Options Index screen 17
- saveds 112, 144
- scoptions file 79
- shortintegers 112, 178, 179, 183
- smallcode 113, 132, 177
- smalldata 113, 132, 177
- solving problems 196
- sourceis= 113
- specifying 79–80
- stackcheck 113, 160, 161
- stackextend 113, 160, 161
- standardio 114
- startup= 114
- strict 114, 181–182, 183
- stringconst 114
- stringmerge 114–115, 132, 177, 180
- stripdebug 116
- structureequivalence 116, 164, 275
- summary of options 80–84
- switch options 79
- to= 116
- trigraph 116
- underscore 116
- unschar 272
- unsignedchar 116, 177
- utilitylibrary 117
- verbose 117, 173
- version 117
- warn= 117–118, 183, 265
- warnvoidreturn 118
- with= 118
- xreference 118
- xreferenceheaders 118
- xreferencesystem 119
- compiler options, setting
 - gadgets for selecting options 17
 - saving settings 17
 - scopts screens 23
 - scopts utility 16–18
 - Shell method 23
 - specifying from screen editor (se) 26
 - specifying on command line 23
 - Workbench method 16–18
- compiling programs 77–78, 171–173
 - See also compiler options
 - See also sc command
 - Build icon 18–19
 - hunks 171–173
 - numbered compiler messages 206–257
 - portability considerations 181–182
 - sc command 24, 77–78
 - screen editor (se) 48
 - Shell method 22–24
 - solving problems 195–197
 - unnumbered compiler messages 204–205
 - Workbench method 18–19
- CON: keywords 138
- Configuration Window, screen editor (se) 57–64
 - Asm modes 61–62
 - Auto-Indent Mode option 62–63
 - Autowrap option 62
 - Backup File Mode option 63
 - Case Sensitive Characters option 63–64
 - Column Display option 63
 - displaying 57–58
 - illustration 58
 - Input processing option 61–62
 - key definitions, displaying 59–60
 - key definitions, setting 60–61
 - Keys 1–4 pull-down menu 58
 - No processing option 61–62
 - Options pull-down menu 58
 - Project pull-down menu 58
 - Prompt Before Undo option 63
 - reusing saved settings 64
 - saving new settings 64
 - Searches Method option 41, 63
 - Tab Expansion Method option 62
 - tab stops, setting 61
- console window 138
- const keyword problems 196–197

- constlibbase compiler option 88
- control characters, entering 46
- Control-\ escape character 46
- conversion of data types 178–179
- Convert Macro to Key option, Project pull-down menu 55
- converting compiler options
 - Aztec C options 281–284
 - Version 5 options to Version 6 286–293
- converting from Version 5 to Version 6 285–294
 - compilation errors 195–196
 - compiler option conversion 286–293
 - installing Version 6 285
 - recompiling code 285
 - sascopts file conversion 285–286
 - specifying Version 6 libraries 294
 - symbol information 286
- Copy option, Block pull-down menu 53
- correcting errors in source file 16, 19, 24
- coverage compiler option 88–89, 271
- cpr (Codeprobe) command 25
- cpr.guide help database 28
- CPU= compiler option 89
- crashing the machine 200
- Create New CLI option, Windows pull-down menu 54
- cres.o startup module
 - creating residentable programs 143–144
 - definition 140
 - saveds compiler option and `__saveds` keyword not allowed 159
- csource= compiler option 78
- `__ctors` array 146, 147
- customizing screen editor (se)
 - See Configuration Window, screen editor (se)
- `__CXBRK` function 274
- `__CXFPE` function 274
- `__CXOVF` stack overflow routine 160

D

- d command
 - See display command
- d option, se command 38
- data= compiler option
 - changing data addressing methods 177
 - changing default data section 157
 - purpose and use 89–90
 - using in overlays 132
- data storage classes 179–180
- data types and sizes 177–179
 - casting results of short integers 179
 - conversion 178–179
 - effect of shortint option 178, 179
 - narrow types in pre-ANSI function declarations 187–188
 - pointer length 178
 - portability considerations 183
 - signed and unsigned objects 177
 - sizes and values 177–178
 - widest operand 178
- data=auto compiler option 90
- data=far compiler option 89, 182
- data=faronly compiler option 90, 161, 182
- datamem=chip compiler option 172
- datamemory= compiler option 90, 177
- dataname= compiler option 91
- data=near compiler option 89, 143, 172
- date and time
 - `__DATE` preprocessor macro 273
 - locale-specific behavior 278–279
- daylight savings time, default 278
- debug= compiler option 91, 173
- Debug icon
 - definition 14
 - starting the debugger 20
- debug=full compiler option 91
- debug=fullflush compiler option 91
- debugging programs
 - cpr (CodeProbe) command 25
 - Dialog window 20–21, 25
 - display command 21
 - go command 21, 25
 - problems with CodeProbe 197
 - quit command 21, 25
 - Source window 20–21, 25
 - stepping through programs 21
 - using Shell 25
 - using Workbench 20–21
- debug=line compiler option 91
- debug=symbol compiler option 91
- debug=symbolflush compiler option 16, 17, 91
- declarators
 - SAS/C Compiler behavior 272
- def_
 - beginning of names for icons 25–26

- default icons for tools 26
- define= compiler option 92
- define linker option
 - eliminating standard I/O window 137
 - forcing symbol name references 286
 - purpose and use 121–122
- Delay function (AmigaDOS) 20
- Delete option, Block pull-down menu 52, 53
- deletetabs.se, AREXX macro 69
- deleting text 41, 52
 - See also undoing changes
- detachable programs 142–143
 - See also residentable programs
 - cback.o startup module 142–143
 - creating 142
 - definition 142
 - external definition requirements 142–143
 - restricting to one copy running 143
- diagnostics
 - SAS/C Compiler behavior 268
- Dialog window (CodeProbe) 20–21, 25
- directories created by installation program
 - 6–8
 - c 6
 - env 7
 - examples 6
 - help 6
 - icons 6
 - Include 7
 - Lib 7
 - libs 7–8
 - source 7
 - starter_project 6
- disassemble= compiler option 92
- diskcopy command (AmigaDOS) 3
- display command 21
- Display Names option, Project pull-down menu 56
- DOSBase
 - initialized by startup code 149
- _dtors array 146, 147

E

- Edit icon 14, 38
- editor
 - See screen editor (se)
- Electronic Mail Interface to Technical Support (EMITS) 31–32

- End option, Block pull-down menu 51, 53
- enumerated types, specifying size 167
- enumerations
 - SAS/C Compiler behavior 271
- env directory 7
- ENV: logical device 10
- ENV:sc subdirectory 11
- environment
 - SAS/C Compiler behavior 268
- environment names 278
- environment variables, setting 10–11
- equivalent structures 164–166
 - definition 164
 - examples 165, 166
- error and warning messages 203–266
 - eliminating informational messages 202
 - enabling suppressed messages 265
 - linker messages 258–264
 - numbered compiler messages 206–257
 - unnumbered compiler messages 204–205
 - using error and warning messages 203–204
- error compiler options
 - error= 92
 - errorconsole 92
 - errorhighlight 93
 - errorlist 93
 - errorrex 16, 17, 93
 - errorsources 93
 - maximumerrors= 102
 - onerror= 107–108
- errors in source file, correcting 16, 19, 24
- escape character (Control-\) 46
- examples directory 6
- executable module 172
- exit function 277
- Exit SE option, Project pull-down menu 47, 56
- exiting screen editor (se) 46–47, 56
- extern BPTR —_Backstdout external
 - definition 143
- external data, common model 168–169
- external storage class 180
- external symbol information in hunks 172
- externaldefs compiler option 93

F

- fancy linker option 122

- far BSS hunk 172
 - far data 172
 - far data hunk 172
 - _far keyword 154, 156–157
 - affecting only calling function 155
 - compatibility with previous releases 157
 - declaring data items 176
 - declaring far data 172
 - overriding data=auto compiler option 90
 - purpose and use 156–157
 - fast linker option 122
 - faster linker option 122
 - fax numbers for Technical Support 30
 - fflush function 275
 - fgetpos function 276
 - filename extensions
 - options for specifying with sc command 78
 - recognized by sc command 78
 - files
 - See also screen editor (se)
 - editing new files 55
 - maximum number of open files 39
 - naming 56
 - opening more than one 38
 - saving 46–47
 - specifying multiple filenames with sc command 78
 - switching between 38, 39
 - truncation after a write 275
 - FindPort function 143
 - findsym.se, AREXX macro 69
 - findsymbol= compiler option 93
 - flibc #pragma statement 162
 - floating-point numbers
 - conversion types 179
 - SAS/C Compiler behavior 270
 - value limits 178
 - floppy disk product installation 8–9
 - disks created 9
 - Pretend to Install process 8–9
 - read.me file 9
 - fmod function 274
 - fopen function 275
 - formal storage class 180
 - formatted print functions
 - %p conversion 275–276
 - problems 197
 - _fpinit function 146
 - _fpinit.c function 270
 - fprint problems 197
 - fprintf function 275
 - _fpterm function
 - from compiler option 94
 - from linker option 122
 - fscanf function 276
 - ftell function 276
 - function pointers
 - effect of keywords 155
 - functions
 - calling functions in overlays 128
 - function call problems 199
 - writing replacement functions for SAS/C
 - library functions 202
 - fwidth linker option 122
- ## G
- g command
 - See go command
 - genprotodataitems compiler option 94
 - genprotoexterns compiler option 94
 - genprotofile= compiler option 94
 - genprotoparameters compiler option 94
 - genprotos compiler option 94–95
 - genprotostatistics compiler option 95
 - genprototypedefs compiler option 95
 - geta4 function 159
 - getch function 198
 - getchar problems 198
 - global message port 143
 - global optimization 159, 180, 271
 - global symbol information in hunks 172
 - globalsymboltable= compiler option 96
 - See also makeglobalsymboltable= compiler option
 - go command
 - completing program execution 21, 25
 - grep
 - compared with screen editor (se)
 - searching 43
 - GST= compiler option 96
 - GSTimmediate compiler option 96
- ## H
- .h filename extension 78

- H_DEBUG hunk 173
- H_SYMBOL hunk 173
- hard disk product installation 4–8
 - directories created 6–8
 - modifying startup files 5–6
 - Pretend to Install process 4
- header files
 - function call problems 199
 - incorrect characters at beginning 196
 - multipleincludes compiler option 105
 - reasons for including incorrect file 199
- heap area 175
- height linker option 122
- help command 27
- help directory 6
- help system 27–29
 - See also problem solving
 - See also Technical Support
 - help databases 28–29
 - installation procedure 4
 - invoking AmigaGuide 28
 - menu bar of help screens 29
 - navigating through help databases 29
 - sc command 78
 - screen editor (se) 48
 - Screen Editor Help screen 28
- hunk_symbol records 121
- hunks 171–173
 - changing hunk names 175
 - chip hunk 172
 - code hunk 172
 - contained in primary (root) node 174
 - contents 171–172
 - far BSS hunk 172
 - far data hunk 172
 - H_DEBUG hunk 173
 - H_SYMBOL hunk 173
 - letters indicating hunks produced by
 - compiler 173
 - near BSS hunk 172
 - near data hunk 172
 - NTRYHUNK 133, 175
 - reserved section names 175
 - udata hunk 172
 - _MERGED hunk 172, 175
 - _NOMERGE hunk 175
- hwidth linker option 123
- I
- icons 25–26
 - contained in starter_project directory 14
 - default icons for tools 26
 - modifying 26
 - naming conventions 25
 - object file icons 26
 - preventing all icons 26
 - preventing specific types 26
 - representing SAS/C tools 14
 - templates in sc:icons drawer 25
- icons compiler option 96–97
- icons directory 6
- identifierlength= compiler option 97
- identifiers
 - SAS/C Compiler behavior 268
- ignore= compiler option 97
- implementation-defined behavior 267–279
 - ANSI definition 267
 - arrays and pointers 270–271
 - bitfields 271
 - characters 269
 - declarators 272
 - enumerations 271
 - environment 268
 - floating-point numbers 270
 - identifiers 268
 - integers 269–270
 - library functions 273–278
 - locale-specific behavior 278–279
 - preprocessing directives 272–273
 - qualifiers 272
 - registers 271
 - statements 272
 - structures 271
 - translation 268
 - unions 271
- implicit structure references 164
- Include directory 7
- #include files search paths 272
- include: device name, assigning 10
- includedirectory= compiler option 97
- incomplete structure tags 188–189
- indent linker option 123
- indent.se, AREXX macro 70
- indirection operator (*) 155
- _inline keyword 159

- Input Processing option 61–62
 - Asm modes 61–62
 - Autowrap 61–62
 - No processing 61–62
- Insert File option, Project pull-down menu 52
- inserting text 40–41, 52
- installing SAS/C Development System 3–11
 - assigning logical device names 10
 - directories created by installation
 - program 6–8
 - floppy disk installation 8–9
 - hard disk installation 4–8
 - modifying startup files 5–6
 - preparing for installation 3
 - read.me file 5
 - setting environment variables 10–11
- integers
 - SAS/C Compiler behavior 268
 - value limits 177–178
- Interlace Toggle option, Windows pull-down menu 54
- _interrupt keyword 155, 157
- IntuiMessage structure 165, 166
- IOExtSer structure 166
- isalnum function 273–274
- isalpha function 273–274
- iscntrl function 273–274
- islower function 273–274
- isprint function 273–274
- isupper function 273–274

K

- keepline compiler option 97
- key definitions 59–61
 - displaying 59–60
 - qualifiers 60
 - raw code 60
 - setting 60–61
- keyboard shortcuts 50
- keys
 - accelerator keys 50
 - Alt keys 60
 - assigning AREXX macros 66
 - block operations 53
 - changing colors 54
 - deleting text 41, 52
 - editing new files 55
 - exiting screen editor (se) 56
 - inserting text 52
 - keystroke macros 55
 - managing windows 54
 - moving around in files 40, 51
 - qualifier keys 59
 - saving files 56
 - searching for and replacing strings 53
 - Shift keys 60
 - undoing changes 56
- Keys pull-down menu
 - definition 58
 - determining functions of commands 59
- keystroke equivalents for AREXX
 - commands 70–73
- keystroke macros 45–46
 - assigning macros to another key 46
 - creating 45
 - keys 55
 - menu options 55
 - reloading 45–46
 - replacing 45
 - saving 45–46
- keyword options 79
- keywords
 - affecting caller and callee 155
 - affecting definition of functions 155
 - affecting only calling function 155
 - /AUTO 138
 - /CLOSE 138
 - CON: keywords 138
 - const 196–197
 - controlling standard I/O window 138
 - list of special keywords 154
 - register 180, 271
 - signed 177, 183
 - specifying on function pointers 155
 - unsigned 177
 - /WAIT 138
 - _aligned 155–156, 161
 - _asm 155, 157–158
 - _chip 90, 156–157, 176
 - _far 90, 155, 156–157, 172, 176
 - _inline 159
 - _interrupt 157
 - _near 90, 155, 156–157, 172, 176
 - _regargs 111, 155, 157–158, 202
 - _saves 112, 144, 145, 155, 158–159
 - _stackext 155, 160–161
 - _stdargs 111, 155, 157–158

L

- lctosc conversion utility 285–286
- Lib directory 7
- .lib filename extension 78
- lib: device name, assigning 10
- libcode compiler option 98, 159
- libent.o module 144
- libfd linker option 123
- libinit.o module 144
- libinitr.o module 144
- libprefix linker option 123–124
- libraries
 - See also shared libraries
 - problems creating resident libraries 201
 - Version 5 and Version 6 library
 - equivalences 294
- library base pointers
 - See constlibbase compiler option
- library bases, initializing
 - See system library bases, initializing
- library= compiler option 78, 98
- .library files
 - library bases 149
- library functions
 - SAS/C Compiler behavior 273–278
- library linker option 124
- library revision numbers 150
- librevision linker option 124
- libs directory 7–8
 - scdebug.library 7
 - scgo.library 7
 - scgst.library 7
 - schi.library 8
 - sckeymap.library 8
 - sclist.library 8
 - scpeep.library 7
 - scspill.library 8
 - sc1.library 7
 - sc2.library 7
- libversion linker option 124
- Line Number option, Search pull-down
 - menu 51
- link compiler option
 - displayed in SAS/C Compiler Options
 - Index screen 17
 - eliminating standard I/O window 137
 - nostdio option 137
 - purpose and use 98
 - running example.c program 16
 - startup modules 141
 - using with sc command 24, 120
- linker (slink)
 - See slink command
- linker options 121–127
 - addsym 121
 - batch 121
 - bufsize 121
 - chip 121
 - define 121–122, 137, 286
 - fancy 122
 - fast 122
 - faster 122
 - from 122
 - fwidht 122
 - height 122
 - hwidth 123
 - indent 123
 - libfd 123
 - libprefix 123–124
 - library 124
 - librevision 124
 - libversion 124
 - map 124
 - maxhunk 124
 - noalvs 124
 - nodebug 125
 - nover 202
 - onedata 125
 - overlay 125
 - overlymgr 125
 - plain 125
 - prelink 125
 - pwidth 125
 - quiet 126
 - root 126
 - smallcode 126
 - smalldata 126
 - stripdebug 126
 - swidht 126
 - to 126
 - verbose 126
 - verify 126
 - width 126
 - with 127
 - xref 127
- linker symbols
 - absolute symbols not used with
 - residentable programs 159
 - at sign (@) preceding 122

linker symbols (*continued*)

- converting from Version 5 to Version 6 286
- created for residentable programs 144
- extra underscores preceding 122
- NEWDATA1 144
- RESBASE 144
- RESLEN 144
- resolving undefined symbols 193–195
 - `_LinkerDB` 144, 158
- `_LinkerDB` symbol 144, 158
- `linkeroptions=` compiler option 98–99, 120
- linking programs 120–127, 173–174
 - See also linker options
 - See also `sc` command
 - See also `slink` command
 - eliminating standard I/O window 137
 - entering `slink` command 121
 - linker messages 258–264
 - producing executable modules 173–174
 - `sc` compared with `slink` 120
 - startup modules 141–144
 - using Build icon 18–19
 - using `sc` 24
 - using Shell 24
- list compiler option 99
- `listfile=` compiler option 99
- `listheaders` compiler option 99
- `listinclude` compiler option 99–100
- `listmacros` compiler option 100
- `listnarrow` compiler option 100
- `listsystem` compiler option 100
- load and stay resident programs
 - See detachable programs
 - See residentable programs
- load files 172
- Load Macros option, Project pull-down menu 55
- locale-specific behavior 278–279
- logical device names
 - ENV: 10
 - ram: 11
- logical device names, assigning 10
 - include: 10
 - lib: 10
 - sc: 10
- long — `_BackGroundIO` external definition 143
- long — `_priority` external definition 142
- long — `_stack` external definition 142

M

macros

- See AREXX macros
- See keystroke macros
- Main Menu key (F2) 50
- `_main` routine
 - converting incoming command to ANSI format 140
 - opening standard I/O window 137
 - raising argument limits 140–141
- `makedir` command (AmigaDOS) 22
- `makeglobalsymboltable=` compiler option 100
 - See also `globalsymboltable=` compiler option
- `malloc` function 141, 277
- `map` compiler option 100
- `map` linker option 124
- `mapfile=` compiler option 100
- `mapjunk` compiler option 101
- `maplib` compiler option 101
- `mapoverlay` compiler option 101
- `mapsymbols` compiler option 101
- `mapxreference` compiler option 101
- `math=` compiler option 102
- math libraries
 - including incorrect library 194
 - unresolved symbol problems 193–194
- `math=coprocessor` compiler option 270
- mathematics functions 274
- `math=ffp` compiler option 102, 270
- `math=ieee` compiler option 102, 270
- `math=standard` compiler option 16, 17, 102
- `math=68881` compiler option 102, 158
- `math=68882` compiler option 102
- MAXARG symbol 140
- `maxjunk` linker option 124
- `maximumerrors=` compiler option 102
- `maximumwarnings=` compiler option 102
- memory options
 - `bssmemory=` 86
 - `codememory=` 87
 - `datamemory=` 90
 - `memorysize=` 103–104
- `memorysize=` compiler option 103–104
 - See also `argumentsize=` compiler option

See also `preprocessorbuffer=` compiler option
 default values for `argumentsize` and `preprocessorbuffer` options 104
 size specifications 103
 menus
 See also specific menus
 abbreviated menus 50
 block operations 53
 deleting text 52
 editing new files 55
 exiting screen editor (se) 56
 inserting text 52
 keystroke macros 55
 managing windows 54
 moving around in files 51
 naming files 56
 saving files 56
 searching for and replacing strings 53
 selecting menu options 50
 undoing changes 56
`__MERGED` hunk 172, 175
 message browser
 See `scmsg` (message browser)
 messages
 See error and warning messages
 modified compiler option 104
 Move option, Block pull-down menu 53
 moving around in files 40, 51
 msg statement
 See `#pragma msg` statement
`multiplecharacterconstants` compiler option 104–105
`multipleincludes` compiler option 105

N

naming files 56
 narrow types in pre-ANSI function
 declarations 187–188
 national characters in variable names 168
 near BSS hunk 172
 near data 172
 near data hunk 172
`__near` keyword 154, 156–157
 affecting only calling function 155
 compatibility with previous releases 157
 declaring data items 176
 declaring near data 172

overriding data=`auto` compiler option 90
 purpose and use 156–157
 nested comments 168
 See also `commentnest` compiler option
`NEWDATA` symbol 144
 newline character (`\n`), `AREXX` macros 65
`\n` (newline character) 65
 No processing option 61–62
`noalvs` linker option 124
`noautoreg` compiler option 180
`nodebug` linker option 125
`__NOMERGE` hunk 175
 non-reentrant programs 176
`nostdio` option 137
`nover` linker option 202
`NTRYHUNK` code hunk 133, 175

O

`.o` filename extension 78
`object=` compiler option 78, 105
 object file icons 26
`objectlibrary=` compiler option 106
`objectname=` compiler option 106
`oldpreprocessor` compiler option 106–107
`onedata` linker option 125
`onerror=` compiler option 107–108
 online help
 See help system
 Open New Window option, Project pull-down menu 55
 Open Window option
 Project pull-down menu 55
 Windows pull-down menu 47, 54
 operands
 conversion types 179
 widest operand 178
 optimize compiler option 108
`optimizealias` compiler option 108
`optimizercomplexity=` compiler option 108
`optimizerdepth=` compiler option 109
`optimizerglobal` compiler option 109
`optimizerinline` compiler option 109
`optimizerinlocal` compiler option 109
`optimizerloop` compiler option 109
`optimizerpeephole` compiler option 109
`optimizerrecurdepth=` compiler option 110

- optimizersize compiler option 110
- optimizertime compiler option 110
- Options pull-down menu
 - Auto-Indent Mode 62–63
 - Backup File Mode 63
 - Case Sensitive Characters 63–64
 - Column Display 63
 - Configuration Window 57
 - definition 58
 - Input Processing 61–62
 - Prompt Before Undo 63
 - Searches Method 63
 - Tab Expansion Method 62
- __OSlibversion external long integer 150
- overlay linker option 125
- overlay manager
 - customizing 133
 - definition 127
- overlay tree
 - asterisks indicating depth 129
 - definition 127
 - examples 128, 130–131, 132
 - terminating with pound sign (#) 129
- overlays 127–133
 - data=near compiler option 132
 - definition 127
 - functions in overlays 128
 - listing filenames 129
 - maximum number 129
 - nodes in executable module 174
 - overlay option 129
 - root node 127, 128–129
 - smallcode compiler option 132
 - smalldata compiler option 132
 - specifying compiler options 132–133
 - stringmerge compiler option 132
 - using multiple shared libraries instead 127
 - with files 129–130
- overlymgr linker option 125
- overwrite mode 41
- __ ovlymgr global symbol 133

P

- .p filename extension 78
- padding structures 183
- parameters= compiler option 110–111
- parameters=both compiler option 110, 158, 202
- parameters=register compiler option 110, 157–158, 202
- parameters=registers option 294
- parameters=stack compiler option 110, 158
- patterns, searching for 43
- perror function messages 276–277
- plain linker option 125
- plus sign (+)
 - continuing lines of filenames for overlays 129
 - separating filenames for overlays 129
 - separating filenames for sc command 78
- pointers
 - declaring 155
 - default length 178
 - SAS/C Compiler behavior 270–271
- portable code, writing 181–189
 - See also implementation-defined behavior
 - compiling with strict or ANSI options 181–182
 - data and type sizes 183
 - defining __STRICT__ ANSI preprocessor symbol 182
 - guidelines for avoiding problems 182–183
 - incomplete structure tags 188–189
 - narrow types in pre-ANSI function declarations 187–188
 - structure size and padding 493
 - writing structures to disk file 184–186
- pound sign (#)
 - terminating overlay tree 129
- #pragma msg statement 162, 162–163
 - error 162
 - example 163
 - ignore 162
 - pop 162
 - push 162
 - warn 162
- #pragma statements 161–163
 - flibcall 162
 - types of #pragma statements 161
 - using in residentable programs 143
- precision= compiler option 111
- prelink linker option 121, 125
- preprocessing
 - comma (,) operator 168
 - oldpreprocessor compiler option 106–107
 - sizeof operator 168

- preprocessing directives
 - SAS/C Compiler behavior 272–273
- preprocessor symbols
 - defined by compiler 119
 - defined by compiler options 120
 - `__STRICT__` `__ANSI` 85, 182
- `preprocessorbuffer=` compiler option 111
- `preprocessoronly` compiler option 111–112
- primary node 174
- Print option, Block pull-down menu 53
- printing messages in the Shell 143
- `__priority` external definition 142
- problem solving 193–202
 - See also Technical Support
 - `asctime` function 200–201
 - CodeProbe problems 197
 - compilation errors 195–197
 - crashing the machine 200
 - eliminating informational messages 202
 - formatted print functions 197–198
 - `getchar` problems 198
 - incorrect results from function calls 199
 - resident programs or libraries 201
 - resolving undefined symbols 193–195
 - standard I/O window 201
 - writing replacement functions 202
- `proceed` command
 - equivalent to Return key 21
- process restrictions using startup modules 141
- processor-specific code
 - See `CPU=` compiler option
- `programname=` compiler option 112
- Project pull-down menu
 - Close Window 47
 - Convert Macro to Key 55
 - definition 58
 - Display Names 56
 - Exit SE 47, 56
 - Insert File 52
 - Load Macros 55
 - Open New Window 55
 - Open Window 55
 - Rename 56
 - Save & Close 19, 24, 47, 56
 - Save & Continue 47, 56
 - Save & Reopen 47, 55, 56
 - Save Macros 55
 - Undo Last Change 56
- projects, setting up 14–15

- Prompt Before Undo option, Options pull-down menu 63
- prototype generation 94–95
- `PutMsg` exec function 165
- `pwidth` linker option 125

Q

- `q` (quit) command 21, 25
- qualifier keys 59, 60
- qualifiers
 - SAS/C Compiler behavior 272
- question mark (?)
 - getting help for `sc` command 78
- quiet linker option 126
- quit command (debugger) 21, 25

R

- ram: logical device 11
- raw code for key definitions 60
- `rawcon` function 198
- Read option, Block pull-down menu 53
- `read.me` file 5, 9
- `realloc` function 277
- recompiling and linking (rebuilding)
 - programs 19, 24
- reentrant and non-reentrant programs 176
- `__regargs` keyword 157–158
 - affecting caller and callee 155
 - purpose and use 157–158
 - using with user-written replacement functions 111, 202
- register keyword 180, 271
- register storage class 180
- registered parameters 294
- registers
 - SAS/C Compiler behavior 271
- regular expressions
 - definition 41
 - entering in searches 43
 - Use Regular Expressions option 63
- relocation records 172
- `remove` function 275
- Rename option, Project pull-down menu 56

- replacing character strings 42–43
 - entering regular expressions 43
 - keys 53
 - menu options 53
 - options 43
 - prompted replacements 42
- RESEBASE symbol 144
- resetoptions compiler option 79, 112
- resident command (AmigaDOS)
 - not used with detachable programs 143
 - using with residentable programs 143
- resident libraries, creating 201
- resident program
 - definition 143
- residentable programs 143–144
 - See also detachable programs
 - creating 143–144
 - cres.o startup module 143, 144
 - debugging with catchres.o startup module 144
 - requirements 144
 - solving problems 201
 - symbols created by slink 144
 - unable to reference absolute symbols 159
- RESLEN symbol 144
- resolving undefined symbols 193–195
- restoring text
 - See undoing changes
- Return key
 - equivalent to proceed command 21
- reusing saved configuration settings 64
- root linker option 126
- root node
 - contents 128–129, 174
 - definition 127
- running programs 174–175
 - heap area 175
 - Shell method 24–25
 - stack area 174
 - Workbench method 19–20

S

- .s filename extension 78
- s:startup-sequence file
 - modifying 5–6
 - solving problems 195
- s:startupII file, modifying 5
- s:user-startup file
 - modifying 5–6
 - solving problems 195
- SAS/C language
 - See Amiga-specific features of SAS/C language
- sascopts file, converting 285–286
- Save & Close option, Project pull-down menu 19, 24, 47, 56
- Save & Continue option, Project pull-down menu 47, 56
- Save & Reopen option, Project pull-down menu 47, 55, 56
- Save Macros option, Project pull-down menu 55
- saves compiler option 112, 144, 158
- _saves keyword 112, 158–159
 - affecting definitions of functions 155
 - Amiga-specific features 158–159
 - applications without startup modules 145
 - compared with saves compiler option 158
 - equivalent to geta4 function 159
 - not used with cres.o and catchres.o startup modules 144, 159
 - purpose and use 112
- saving
 - compiler options 17
 - files 46–47, 56
 - keystroke macros 45–46
 - new settings in Configuration Window 64
- sc command 77–78
 - See also compiler options
 - compared with slink command 120
 - compiling and linking programs 24
 - default icons 26
 - filename extensions 78
 - link compiler option 24, 120
 - options for specifying filename extensions 78
 - specifying multiple filenames 78
 - specifying startup modules 141
 - startup option 141
 - syntax 77
- sc.guide help database 29
- sc_lib.guide help database 29
- sc_util.guide help database 28
- sc: device name, assigning 10

- scdebug.library 7
- scgo.library 7
- scgst.library 7
- schl.library 8
- sckeymap.library 8
- scslst.library 8
- scmsg (message browser)
 - correcting source file errors 18, 24
 - SAS/C Message Browser window 18
 - specifying errorrexx compiler option 16, 17
- scmsg.guide help database 28
- scoptions file
 - converting sasopts file 285–286
 - overriding settings 79
 - saving options 16, 23
- SOptions icon 14
- scopts utility
 - gadgets for selecting options 17
 - running from screen editor (se) 48
 - setting compiler options 16–18
 - specifying options on command line 23
 - starting with SOptions icon 14, 16
 - using scopts screens 23
- scpeep.library 7
- screen editor (se) 37–48
 - See also AREXX interface
 - See also se command
 - block operations 43–44, 53
 - compiling programs 48
 - control characters 46
 - creating source files 15–16, 22
 - deleting text 41, 52
 - Edit window, illustration 19
 - editing text 40–46
 - entering text 40–46
 - exiting 46–47, 56
 - getting help 48
 - insert mode 40–41, 52
 - keystroke macros 45–46, 55
 - maximum number of open files 39
 - moving around in files 40, 51
 - overwrite mode 41
 - replacing character strings 42–43, 53
 - saving files 46–47, 56
 - Screen Editor Help screen 28
 - searching for character strings 41–42, 43, 53
 - split-screen mode 48
 - starting 14, 38–39
 - status line 39–40
 - substituting another editor 14
 - switching between windows 38, 48
 - undoing changes 44, 56
 - window display size 39
 - window management 47–48, 54
- screen editor (se), controlling 49–56
 - block operations 43–44, 53
 - changing colors 54
 - deleting text 41, 52
 - editing new files 55
 - exiting the editor 47, 56
 - inserting text 40–41, 52
 - keystroke macros 45–46, 55
 - managing windows 47–48, 54
 - moving around in files 40, 51
 - naming files 56
 - replacing character strings 42–43, 53
 - saving files 46–47, 56
 - searching for character strings 41–42, 43, 53
 - selecting menu options 50
 - undoing changes 44, 56
- screen editor (se), customizing
 - See Configuration Window, screen editor (se)
- screen editor (se), starting 38–39
 - se command 38
 - specifying more than one filename 38
 - using Edit icon 14, 38
- scsetup command
 - default icons 26
 - setting up projects 14–15
- scspill.library 8
- sc1.library 7
- sc2.library 7
- sc5 command
 - converting old code to Version 6 285
- se command
 - See also screen editor (se)
 - d option 38
 - default icons 26
 - syntax 38
 - v option 38
- se.dat file 38
- se.guide help database 28
- Search pull-down menu
 - Line Number 51
 - Search Again 53
 - Search and Replace 53
 - String Search 53

- Searches Method option
 - Options pull-down menu 63
 - String Matching 41, 63
 - Use Regular Expressions 63
- searching for character strings 41–42
 - See also replacing character strings
 - backward searching 42
 - case sensitivity 42
 - entering regular expressions 43
 - forward searching 41
 - keys 53
 - menu options 53
 - regular expressions 41
 - String Matching option 41
- semicolon (;)
 - Asm mode 62
- setbuf function 275
- setbufv function 275
- setenv command 11
- shared libraries
 - creating 144
 - using instead of overlays 127
- Shell
 - environment for argv and argc 135–136
 - printing messages in the Shell 143
 - programs linked with startup module 141
 - using SAS/C tools 22–25
- Shift keys
 - equivalence 60
 - independent operation 60
- short integers
 - problems with casting results 179
- shortintegers compiler option 112, 178, 179, 183
- SIGILL signal 274
- SIGINT signal 274
- signal function 274
- signed data types
 - See data types and sizes
- signed keyword 177, 183
- sizeof operator
 - portability of structures 183
 - preprocessor directives 168
- slash (/)
 - comment start sequence (/*) 168
 - slashes (//) defining comment blocks 168
- slink command
 - See also linker options
 - See also linking programs
 - compared with sc command 120
 - default icons 26
 - define linker option 137
 - eliminating standard I/O window 137
 - invoked by smake utility 19
 - linker messages 258–264
 - specifying startup modules 141
 - symbols created for residentable programs 144
 - syntax 121
- smake utility
 - compiling programs 18–19
 - definition 14
 - starting with Build icon 14
- smallcode compiler option 113, 132, 177
- smallcode linker option 126
- smalldata compiler option 113, 132, 177
- smalldata linker option 126
- snapshot files
 - creating 142, 144
 - hard drive corruption 142, 144
- solving problems
 - See problem solving
- source code
 - not placed in sc directory tree 8
- source directory 7
- source file errors, correcting 16, 19, 24
- source files, creating
 - screen editor (se) 15–16, 22
 - Shell 22
 - Workbench 15–16
- Source window (CodeProbe) 20–21, 25
- sourceis= compiler option 113
- special keywords
 - See keywords
- stack
 - alignment problems 156
 - definition 160
 - managing stack space 160–161
 - purpose and use 174
 - running out of stack space 161
 - __stack external definition 142
 - __stack external long integer 160, 161
 - stack overflow routine (—CXOVF) 160
 - stackcheck compiler option 113, 160, 161
 - __stackext keyword 160–161
 - affecting definitions of functions 155
 - managing stack space 160–161
 - pointer register variables 161

- stackextend compiler option 113
 - managing stack space 160, 161
 - pointer register variables 161
 - purpose and use 113
- standard I/O window
 - changing window attributes 137–138
 - declaring `__stdiowin` external variable 137
 - eliminating 137, 201
 - keeping open 20
 - keywords for controlling 138
 - program crashes caused by eliminating 138
- standard programs, creating 142
- standardio compiler option 114
- starter_project directory 6, 14
- startup= compiler option 114
- startup files, modifying
 - adding assign statements 10
 - s:startup-sequence file 5–6
 - s:startupII 5
 - s:user-startup file 5–6
- startup modules 140–145
 - See also autoinitialization function
 - See also autotermination function
 - argument limit, changing 140–141
 - detachable programs (load and stay resident) 142–143
 - linking with startup modules 141–144
 - list of modules 140
 - modifying `__main` 140–141
 - residentable programs 143–144
 - shared libraries 144
 - Shell process restrictions 141
 - standard programs 142
 - tasks performed by all modules 140
 - writing applications without startup modules 145
- startup option 141
- statements
 - SAS/C Compiler behavior 272
- static storage class 180
- status line, screen editor (se) 39–40
- `__STD` preceding function name 145
- `__stdargs` keyword 111
 - affecting caller and callee 155
 - purpose and use 111, 157–158
- `__stdiov37` external variable 138
- `__stdiowin` external variable, declaring 137–138
- stepping through programs 21
- `__STI` preceding function name 145
- `__STKNEED` external long integer 160, 161
- storage classes 179–180
 - automatic 180
 - external 180
 - formal 180
 - register 180
 - static 180
- strerror function 278
- `__STRICT__` ANSI preprocessor symbol 85, 182
- strict compiler option 114, 181–182, 183
- strict reference-definition model 169
- String Search option, Search pull-down menu 53
- stringconst compiler option 114
- stringmerge compiler option 114–115
 - declaring data items 177
 - effect on static objects 180
 - purpose and use 114–115
 - using overlays 132
- stripdebug compiler option 116
- stripdebug linker option 126
- struct IntuiMessage 165, 166
- struct Message 166
- struct WBStartup 136
- structureequivalence compiler option 116, 164, 275
- structures
 - determining size and padding 183–184
 - equivalent structures 164–166
 - implicit structure references 164
 - incomplete structure tags 188–189
 - IntuiMessage structure 165, 166
 - IOExtSer 166
 - portability guidelines 185–186
 - SAS/C Compiler behavior 271
 - writing to disk file 184–186
- `__stub` library function 121
- suppressed messages, enabling 265
- swidth linker option 126
- switch options 79
- Switch Windows option, Windows pull-down menu 39
- Switch windows option, Windows pull-down menu 54
- switching between windows 38, 39, 48
- symbols
 - See linker symbols

SYSBase

- initialized by startup code 149
- system function 278
- system library base pointers
 - See constlibbase compiler option
- system library bases, initializing 147–151
 - automatically initialized library bases 148–149
 - defining library bases in your code 149
 - examples 148, 149–150
 - failure of libraries to open 150–151
 - library bases for .library files 149
 - library revision numbers 150

T

- Tab Expansion Method option, Options pull-down menu 62
- tab stops, setting 61
- tb utility 142, 144
- Technical Support 29–34
 - See also problem solving
 - BIX support 31
 - contacting 29–34
 - EMITS (Electronic Mail Interface to Technical Support) 31–32
 - information required 33–34
 - Internet support 31–32
 - phone number requirement 33
 - phone, mail, and fax support 30
 - problem description requirement 33–34
 - registration number requirement 3, 33
 - return address requirement 33
 - Usenet support 31–32
 - version number requirement 33
 - when to call 29–30
- telephone numbers for Technical Support 30
- text hunk 172
- time and date
 - locale-specific behavior 278–279
- __TIME preprocessor macro 273
- time zone, default 278
- __tinymain routine 290
- to= compiler option 116
- to linker option 126
- Toggle Display Size option, Windows pull-down menu 39, 54
- Toggle Screen Size option, Windows pull-down menu 48

- tolower.se, AREXX macro 70
- toupper.se, AREXX macro 70
- translation (diagnostics)
 - SAS/C Compiler behavior 268
- trigraph compiler option 116

U

- udata hunk 172
- undefined symbols, resolving 193–195
- underscore (_)
 - default prefix for functions 123
 - preceding linker symbols 122
- underscore compiler option 116
- undo command 44
- Undo Last Change option, Project pull-down menu 56
- undoing changes 44, 56
 - fifty line limit 44
 - keys 56
 - last line of text deleted 41
 - menu options 56
 - Prompt Before Undo option 63
- unindent.se, AREXX macro 70
- unions
 - SAS/C Compiler behavior 271
- unnamed unions 163–164
- unschar compiler option 272
- unsigned data types
 - See data types and sizes
- unsigned keyword 177
- unsignedchar compiler option 116, 177
- utilitylibrary compiler option 117

V

- v option, se command 38
- variables
 - displaying values 21
 - national characters in variable names 168
- verbose compiler option 117, 173
- verbose linker option 126
- verify linker option 126
- version compiler option 117
- Version 5, converting
 - See converting from Version 5 to Version 6

W

- /WAIT keyword 138
- warn= compiler option
 - enabling suppressed messages 265
 - enabling warnings 183
 - purpose and use 117–118
- warnings, enabling
 - See also error and warning messages
 - See also warn= compiler option
 - incomplete structure tags 189
 - narrow types 188
- warnvoidreturn compiler option 118
- widest operand 178
- width linker option 126
- windows
 - displaying two files 39
 - managing in screen editor (se) 47–48, 54
 - switching between windows in screen editor (se) 38, 39
 - text window size in screen editor (se) 39
- Windows pull-down menu
 - Close Window 54
 - Create New CLI 54
 - Interlace Toggle 54
 - Open Window 47, 54
 - Switch Windows 39
 - Switch windows 54
 - Toggle Display Size 39, 54
 - Toggle Screen Size 48
- with= compiler option 118
- with files
 - asterisk (*) indicating depth of overlay tree 129
 - examples 130–131
 - format 129
 - plus sign (+) separating filenames 129
 - pound sign (#) terminating overlay tree 129
- with linker option 127
- Workbench 135–138
 - environment for argv and argc 136
 - program structure, example 136
 - running programs 135–138
 - standard I/O window management 137–138
 - using SAS/C tools 14–21
- Write option, Block pull-down menu 53
- writing portable code
 - See portable code, writing

X

- __XCEXIT function 151
- XDEFS in hunks 172
- xref linker option 127
- xreference compiler option 118
- xreferenceheaders compiler option 118
- xreferencesystem compiler option 119

Z

- zero-length arrays 163–164

Special Characters

- { (braces)
 - See braces ({ })
- \ (backslash)
 - See backslash (\)
- / (slash)
 - See slash (/)
- + (plus sign)
 - See plus sign (+)
- * (asterisk)
 - See asterisk (*)
- *__procname external definition 142
- ; (semicolon)
 - See semicolon (;)
- , (comma)
 - See comma (,)
- _(underscore)
 - See underscore (_)
- ? (question mark)
 - See question mark (?)
- : (colon)
 - See colon (:)
- # (pound sign)
 - See pound sign (#)
- @ (at sign)
 - See at sign (@)

1

- 16-bit address
 - advantages and disadvantages 176
 - relative to program counter 176
 - relative to register A4 176

316 *Index*

16-bit references

- specifying with code= compiler option
87
- __far keyword 156
- __near keyword 156

3

32-bit address

- advantages and disadvantages 176

32-bit references

- chip data 176
- specifying with code= compiler option
87

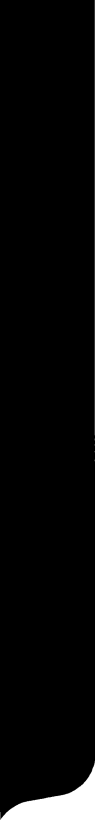
Your Turn

If you have comments or suggestions about the *SAS/C Development System User's Guide, Volume 1: Introduction, Compiler, Editor, Version 6.0* or the SAS/C Development System, please send them to us on a photocopy of this page.

Please return the photocopy to the Publications Division (for comments about this book) or the Technical Support Division (for suggestions about the SAS/C Development System) at SAS Institute Inc., SAS Campus Drive, Cary, NC 27513.

SAS/C[®] Development System
User's Guide, Volume 2:
Debugger, Utilities, Assembler,
Version 6.0

SAS/C[®] Development System User's Guide, Volume 2:
Debugger, Utilities, Assembler, Version 6.0



■ **SAS/C[®] Development System User's Guide**

Volume 2:

Debugger, Utilities, Assembler

Version 6.0



SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/C* Development System User's Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0*, Cary, NC: SAS Institute Inc., 1992. 363 pp.

SAS/C* Development System User's Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0

Copyright © 1992 by SAS Institute Inc., Cary, NC, USA.
ISBN 1-55544-497-0

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.
1st printing, June 1992

The SAS® System is an integrated system of software providing complete control over data access, management, analysis, and presentation. Base SAS software is the foundation of the SAS System. Products within the SAS System include SAS/ACCESS®, SAS/AF®, SAS/ASSIST®, SAS/CPE®, SAS/DMI®, SAS/ETS®, SAS/FSP®, SAS/GRAPH®, SAS/IML®, SAS/IMS-DL/I®, SAS/OR®, SAS/QC®, SAS/REPLAY-CICS®, SAS/SHARE®, SAS/STAT®, SAS/CALC®, SAS/CONNECT®, SAS/DB2®, SAS/EIS®, SAS/ENGLISH®, SAS/INSIGHT®, SAS/LAB®, SAS/LOOKUP®, SAS/NVISION®, SAS/PH-Clinical®, SAS/SQL-DS®, SAS/TOOLKIT®, and SAS/TUTOR® software. Other SAS Institute products are SYSTEM 2000® Data Management Software, with basic SYSTEM 2000, CREATE®, Multi-User®, QueX®, Screen Writer®, and CICS interface software; NeoVisuals® software; JMP®, JMP IN®, and JMP Serve® software; SAS/RTERM® software; and the SAS/C® Compiler and the SAS/CX® Compiler. MultiVendor Architecture™ and MVA™ are trademarks of SAS Institute Inc. SAS Institute also offers SAS Consulting® and On-Site Ambassador™ services. SAS Communications®, SAS Training®, SAS Views®, the SASware Ballot®, and Observations™ are published by SAS Institute Inc. All trademarks above are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. The Institute is a private company devoted to the support and further development of its software and related services.

Amiga Includes and Development Tools Version 2.0

Copyright © 1990 Commodore-Amiga, Inc. All rights reserved.

Amiga Workbench™ Version 2.0

Copyright © 1988, 1990 Commodore-Amiga, Inc. All rights reserved.

Installer

Copyright© 1991-1992 Commodore-Amiga, Inc. All rights reserved.

AmigaGuide™ and WDisplay

Copyright© 1991-1992 Commodore-Amiga, Inc. All rights reserved.

Distributed under license from Commodore.

Amiga®, AmigaDOS®, Workbench®, and AmigaGuide™ are registered trademarks or trademarks of Commodore-Amiga, Inc. Commodore® is a registered trademark or trademark of Commodore Electronics Limited.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Contents

vii Reference Aids

ix Credits

xi Acknowledgments

xiii Using This Book

Part 1 Using the SAS/C Debugger

Page	3	Chapter 1 Introduction to CodeProbe
	3	Introduction
	4	CodeProbe Overview
	9	Compiling and Linking Your Program
	9	Running CodeProbe
	12	CodeProbe Windowing Interface
	16	Entering Commands
	25	Chapter 2 Sample CodeProbe Session
	25	Introduction
	25	A Walk through a Typical Application
	35	Chapter 3 Debugging Your Program
	35	Introduction
	37	Sample Program 1
	39	Controlling Program Execution
	53	Sample Program 2
	57	Examining Data and Data Structures
	73	Chapter 4 Setting Up the Debugging Environment
	73	Introduction
	77	Determining Source File Location
	78	Choosing Command-line Options
	81	CodeProbe Initialization
	82	Setting and Showing Options within CodeProbe
	89	Customizing Dialog Window Commands
	95	Chapter 5 Working in Cross-Development Mode
	95	Introduction
	95	Why Debug in Cross-Development Mode?
	96	Using the Cross Debugger

101 Chapter 6 Using AREXX Macros with CodeProbe

- 101 The AREXX Interface
- 101 Macros Provided by CodeProbe
- 102 Invoking Macros
- 103 Values Returned from CodeProbe Commands

105 Chapter 7 Debugging Resident Libraries

- 105 Introduction
- 105 Setting Breakpoints
- 106 The symload Command
- 106 The libraries.cpr AREXX Macro
- 106 Cautions
- 106 Limitations
- 106 Example

107 Chapter 8 Debugging Multitasking Programs

- 107 Introduction
- 108 Types of Tasks
- 108 How CodeProbe Handles Tasks
- 109 Commands Used to Control Tasks
- 113 Design Considerations for Debugger Compatibility

115 Chapter 9 Commands and Built-in Functions

- 115 Introduction
- 115 CodeProbe Commands
- 118 Special Parameters
- 126 Command Reference
- 202 Built-in Functions
- 203 Built-in Function Reference

Part 2 Using the SAS/C Utilities

215 Chapter 10 Utility Reference

Part 3 Using the SAS/C Macro Assembler

299 Chapter 11 Using Assembly Language with C Language

- 299 Introduction
- 300 Writing Assembly Language Functions
- 314 Communicating between Assembly Language and C Language
- 323 Running the Assembler

Part 4 Using the SAS/C Optimizer

329 Chapter 12 Optimizing Your Code

329 Introduction

329 The Global Optimizer

334 The Peephole Optimizer

334 Running the Optimizers

Part 5 Appendix

341 Appendix diff File-Matching Algorithm

343 Index

Reference Aids

Displays

- 12 1.1 Initial CodeProbe Screen
- 13 1.2 CodeProbe Windows
- 28 2.1 Source Window Showing Mixed Mode
- 28 2.2 Source Window Showing Assembly Mode

Figures

- 275 10.1 Example File Dependencies
- 317 11.1 The Stack after the func Module Is Called

Tables

- 17 1.1 Command Line Editing Functions
- 18 1.2 Escape Sequences in Character Strings
- 19 1.3 Escape Sequences in Commands
- 20 1.4 Pull-down Menu Items
- 23 1.5 Function and Special Keys
- 24 1.6 Menu Accelerator Keys

Credits

Software

The SAS/C Development System is designed, built, and tested through the combined effort of many people at SAS Institute. The AmigaDOS development group includes the following individuals:

Principal Developers	Steve Krueger, Douglas Walker, Michael S. Whitcher
Testing and Technical Support	Twilah K. Blevins, Jim Cooper, Bob Patten, Khristi Tomlinson

These individuals continue to support future development of the SAS/C Development System.
Others who have contributed to the SAS/C Development System are as follows:

Development	Edmund Burnette, Dave Frederick, Russell Gonsalves, Tony Hefner, Joe Hutchinson, Glenn Musial, John Roth
Testing and Technical Support	Alan R. Beale, John W. Gillikin, Charles Tudor

Documentation

Composition	Candy R. Farrell, Denise T. Jones, Pamela A. Troutman
Graphic Design	Creative Services Department
Proofreading	Patsy P. Blessis, Heather B. Dees, Laurie J. Medley, Karen A. Olander, Josephine P. Pope, Christian Schwoerke, Susan E. Willard
Technical Review	Elizabeth Brownrigg, Edmund Burnette, Jim Cooper, Steve Krueger, John Roth, Khristi Tomlinson, Douglas Walker, Michael S. Whitcher
Writing and Editing	Jeanne Ferneyhough, Hanna Hicks, Keith Wagner, Steve Krueger, Douglas Walker, Helen Weeks, Khristi Tomlinson

Acknowledgments

Many people make significant and continuing contributions to the development of the SAS/C Development System. First among them is John A. Toebes VIII, who was instrumental in the development of Versions 4.0 and 5.0 and who directed the early development of Version 6.0.

Others who have contributed to the development and testing of Version 6.0 of the SAS/C Development System include Ralph Babel, Bruce M. Drake, Nathan S. Fish, Maximilian Hantsch, Randell Jesup, David N. Junod, Andrew Kopp, Martin J. Laubach, Christian Ludwig, Andrew N. Mercier, Mike Meyer, Larry Rosenman, Carolyn Scheppner, Steve Shireman, Michael Sinz, Martin Taillefer, Steve Tibbett, John Wiederhirm, Loren Wilton, and Kenneth Yarnall.

The final responsibility for the SAS/C Development System lies with SAS Institute alone. We hope that you will always let us know your feelings about our product and its documentation. It is through such communication that the progress of SAS software continues to be accomplished.

Using This Book

Purpose

SAS/C Development System User's Guide, Volume II: Debugger, Utilities, Assembler describes how to use some of the major tools available to the SAS/C programmer under AmigaDOS: the debugger, the macro assembler, the optimizer, and the various utilities that are part of the SAS/C Development System.

“Using This Book” describes how you can best use this book. It describes the book’s intended audience, the audience’s prerequisite knowledge, the book’s organization and its conventions, and additional documentation that is available to you.

Audience

The information in this book is written for experienced C programmers. This book assumes you have a working knowledge of the C language and the AmigaDOS operating system. To use the SAS/C Macro Assembler, you also need a working knowledge of assembly language. For a list of publications you can use to learn the C language or the AmigaDOS operating system, refer to the “Additional Documentation” section at the end of “Using This Book.”

How to Use This Book

This section gives an overview of the book’s organization and content. The book’s chapters are described, followed by a section on how to use each chapter.

Organization *SAS/C Development System User's Guide, Volume II: Debugger, Utilities, Assembler* is divided into four parts plus an appendix; the chapters contained in each part and the appendix are described here:

Part 1: Using the SAS/C Debugger

Part 1 contains nine chapters that discuss CodeProbe and how to use it to debug your program.

Chapter 1, “Introduction to CodeProbe”

explains what CodeProbe does, how it works, and what tasks are involved in debugging a program. This chapter also describes how to compile and link your program under CodeProbe control.

Chapter 2, “Sample CodeProbe Session”

contains a brief sample session using the `lines.c` program in the `sc:examples` drawer. You can follow this sample session to gain experience with the most commonly used debugger commands.

Chapter 3, “Debugging Your Program”

describes the CodeProbe commands most frequently used for stepping through your program, displaying code and data, and modifying code or data.

Chapter 4, “Setting Up the Debugging Environment”

describes the commands and options you can use to customize the environment in which you debug your program.

Chapter 5, “Working in Cross-Development Mode”

describes how to run your executable code from a serially-connected Amiga system.

Chapter 6, “Using AREXX Macros with CodeProbe”

describes how to use AREXX macros from inside CodeProbe. This chapter contains a list of valid AREXX commands.

Chapter 7, “Debugging Resident Libraries”

describes how to use CodeProbe to debug resident libraries.

Chapter 8, “Debugging Multitasking Programs”

describes how to use CodeProbe with applications that spawn additional tasks.

Chapter 9, “Commands and Built-In Functions”

describes each CodeProbe command and built-in function in detail.

Part 2: Using the SAS/C Utilities

Part 2 contains only one chapter. This chapter discusses the SAS/C utilities.

Chapter 10, “Utility Reference”

describes each of the utilities (such as `smake`, `scopts`, and `tb`) provided by the SAS/C Development System.

Part 3: Using the SAS/C Macro Assembler

Part 3 contains only one chapter. This chapter discusses how to use assembly language with C language.

Chapter 11, “Using Assembly Language with C Language”

describes how to communicate between C language and the assembler. You can incorporate assembler routines as part of your C program or call C functions from assembly language.

Part 4: Using the SAS/C Optimizer

Part 4 contains only one chapter. This chapter discusses how to use the SAS/C Optimizer.

Chapter 12, “Optimizing Your Code” describes what the global and peephole optimizers do and how to run the optimizers during compilation.

The appendix, “diff File-Matching Algorithm,” describes the algorithm used by the `diff` utility to compare files.

What You Should Read

The following table points you to specific chapters in this book for information on particular topics. For other references, refer to “Additional Documentation” later in this section.

For information on	You should read
compiling, linking, and running your program under the debugger	Chapters 1 and 4
using the most common debugger commands	Chapters 2 and 3
customizing your debugging environment	Chapter 4
running your program on one machine and debugging it on another machine	Chapter 5
using AREXX macros	Chapter 6
debugging resident libraries	Chapter 7
debugging multitasking programs	Chapter 8

Conventions

This section covers the typographical and syntax conventions used in this book.

Typographical Conventions

The SAS/C books for AmigaDOS use special fonts to depict specific types of information. These typographical conventions are as follows:

- `roman` is the basic type style used for most text.
- `monospace` is used to show example statements or programs. Monospace is used also for items specific to the

C language, such as the names of functions, header files, and keywords.

oblique is used for arguments or variables whose values are supplied by the user; for example, you should enter an appropriate filename when you see *filename*.

italic is used for terms that are defined. Italic is also used to indicate arguments for which you supply a value.

Syntax Conventions

This book uses the following conventions for syntax:

- monospace** indicates commands, keywords, and switches that should be spelled exactly as shown. These arguments may or may not be optional, depending on whether they are enclosed in square brackets.
- italic* indicates arguments for which you supply a value.
- []** (square brackets) indicate an optional argument when they surround the argument.
- ...** (ellipsis) indicates that you can repeat the argument or group of arguments preceding the ellipsis any number of times.
- |** (vertical bar) means to choose one item from a group of items separated by the bars.

The following example illustrates these syntax conventions:

env [*function* | *integer*]

env

is a command name, so it appears in monospace type.

function

is a function for which you supply the name, so it appears in italic type.

[*function* | *integer*]

are both optional, so they are enclosed in square brackets.

function | *integer*

indicates that you can specify only one of the items separated by the vertical bar.

Additional Documentation

The following sections list documentation you may find helpful in using the SAS/C Development System.

SAS Documentation

If you are interested in SAS documentation, you need to contact the Book Sales department by writing to the following address or by calling the following number:

SAS Institute Inc
Book Sales Department
SAS Campus Drive
Cary, NC 27513
919-677-8000

SAS/C Development System

This book is part of a set of publications for the SAS/C Development System. There are three other publications in the set:

- ☐ *SAS/C Development System User's Guide, Volume I: Introduction, Compiler, Editor* describes how to
 - ☐ install the SAS/C Development System on your Amiga system
 - ☐ use your development environment
 - ☐ create files in the editor
 - ☐ compile and link C files.
- ☐ *SAS/C Development System Library Reference* describes how to
 - ☐ access libraries
 - ☐ create your own libraries
 - ☐ use header files to reduce the amount of time and space required by your program.
- ☐ *SAS/C Development System Quick Reference, Version 6.0* contains reference tables for the library functions, compiler and linker options, and debugger commands.

These volumes are sold together as a set. You can order them by specifying order #A56185.

Other Documentation

The following sections list additional reference material specific to the C language, Amiga and AmigaDOS programming, and the Motorola 68xxx microprocessor.

C Language

The following books describe the C programming language:

American National Standards Committee (1990), *American National Standard for Information Systems—Programming Language C*, Document Number X3J11/90-013, Washington, D.C.: X3 Secretariat: Computer and Business Equipment Manufacturers Association.

Harbison, Samuel P. and Steele, Guy L., Jr. (1990), *C: A Reference Manual, Third Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Kernighan, Brian W. and Ritchie, Dennis M. (1988), *The C Programming Language, Second Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Amiga and AmigaDOS

The following books provide information specifically about programming on the Amiga system. Most of the examples in these books are written in the C or assembler languages.

Commodore-Amiga, Inc. (1991), *Amiga Hardware Reference Manual, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.

Commodore-Amiga, Inc. (1991), *Amiga ROM Kernal Reference Manual: Devices, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.

Commodore-Amiga, Inc. (1991), *Amiga ROM Kernal Reference Manual: Includes and Autodocs, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.

Commodore-Amiga, Inc. (1991), *Amiga User Interface Style Guide*, Reading, MA: Addison-Wesley Publishing Company.

Commodore-Amiga, Inc. (1991), *The AmigaDOS Manual, 3rd Edition*, New York, NY: Bantam Books.

Commodore-Amiga, Inc. (1992), *Amiga ROM Kernal Reference Manual: Libraries, 3rd Edition*, Reading, MA: Addison-Wesley Publishing Company.

Motorola 68xxx The following books contain information specifically about programming the Motorola 68xxx microprocessor.

Motorola, Inc. (1987), *MC68881/MC68882 Floating-Point Coprocessor User's Manual, First Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Motorola, Inc. (1989), *MC68030 Enhanced 32-Bit Microprocessor User's Manual, Second Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Motorola, Inc. (1989), *Programmer's Reference Manual*, Phoenix, AZ: Motorola Literature Distribution.

**Contacting
CATS** You can contact Commodore Application and Technical Support at the following address:

1200 Wilson Drive
West Chester, PA 19380
215-429-0643

Part 1 **Using the SAS/C[®] Debugger**

Chapters	1	Introduction to CodeProbe
	2	Sample CodeProbe Session
	3	Debugging Your Program
	4	Setting Up the Debugging Environment
	5	Working in Cross-Development Mode
	6	Using AREXX Macros with CodeProbe
	7	Debugging Resident Libraries
	8	Debugging Multitasking Programs
	9	Commands and Built-in Functions

1 Introduction to CodeProbe

3	<i>Introduction</i>
4	<i>CodeProbe Overview</i>
4	<i>Basic Features</i>
4	<i>Multitasking Features</i>
5	<i>Summary of Changes and Enhancements for Version 6.0 of the CodeProbe Debugger</i>
5	<i>Terminology</i>
7	<i>How CodeProbe Works</i>
9	<i>Compiling and Linking Your Program</i>
9	<i>Running CodeProbe</i>
9	<i>The cpr Command</i>
10	<i>The quit Command</i>
10	<i>Workbench Support</i>
11	<i>Line Mode</i>
12	<i>CodeProbe Windowing Interface</i>
12	<i>Screen Layout</i>
14	<i>CodeProbe Windows</i>
16	<i>Entering Commands</i>
16	<i>Command Line</i>
20	<i>Pull-Down Menus</i>
22	<i>Function and Special Keys</i>
23	<i>Menu Accelerator Keys</i>

Introduction

CodeProbe* is a powerful source-level debugger that enables you to monitor, line by line, the behavior of programs written in C language, assembly language, or a combination of the two. CodeProbe has many useful features and commands designed to aid you in tracking down your program's problems. Some of these aids provide the ability to examine and modify variables, address and data registers, and user-specified memory locations. CodeProbe gives you absolute control over the execution of your program by enabling you to stop execution at any point. It also enables you to examine both the C source and the

* Throughout this manual, the terms CodeProbe and the debugger are used interchangeably to refer to this debugger program.

assembler generated for your program at the same time so you can see what is really occurring at the assembly language level.

This chapter explains what CodeProbe does, how it works, and what is involved in debugging a program.

CodeProbe Overview

CodeProbe implements a number of commands and features specifically designed to ease the debugging of C programs. In addition to a command-line interface, the debugger provides you with a sophisticated, screen-oriented interface where you can take advantage of multiple windows and pull-down menus. Shortcuts that are built into the program enable you to issue commands with a couple of keystrokes or mouse clicks.

Basic Features As a source-level debugger, CodeProbe allows you to

- ☐ single-step through programs either in C source code or the generated assembler code
- ☐ set breakpoints on C source lines, or an individual assembly language statement
- ☐ continue program execution until a specified function or line number is reached, or until a variable contains a specified value or its value changes
- ☐ examine variables and code at the C language level
- ☐ display C data types—including structures, unions, arrays, enums, typedefs, and bit fields—in their C formats
- ☐ display memory dumps in a variety of formats
- ☐ continuously watch any C variable, array element, or structure member
- ☐ assign a value to a C variable
- ☐ fill an area of memory
- ☐ copy one C array or structure to another
- ☐ copy a string from one place to another
- ☐ view and manipulate register data
- ☐ access storage classes of data, including automatic, static, and register as well as external.

Multitasking Features CodeProbe allows you to debug multitasking programs. Any new tasks or processes created by a program that is being debugged are automatically under debugger control. The following multitasking features are provided:

- ☐ Breakpoints can be set and trapped for any task generated under debugger control.

- The state of any task in a multitasking application can be displayed including the stack and registers.
- Tasks can be started and stopped selectively.
- Tasks that are executing independently of the debugger can be intercepted and attached to the debugger. This can be useful if a process enters an infinite loop or waits indefinitely on a message port.

Summary of Changes and Enhancements for Version 6.0 of the CodeProbe Debugger

The following changes and enhancements are introduced with Version 6.0 of the CodeProbe debugger:

- Cross debugger support is provided via the serial port or a communications network.
- Debugging of shared libraries and devices has been simplified.
- C expression parsing is allowed with most debugger commands.
- The ability to display ranges within arrays is provided.
- The **alias** and **define** commands provide greater flexibility.
- 68881 floating-point registers are supported.
- Motorola fast floating-point format (ffp) display is provided.
- The **env** and **where** commands enable you to look back up the function call chain.
- The **args** command displays arguments to the current function.
- More compact debugging format saves space on disk.
- The AmigaDos Version 2.0 look and feel is provided, including public screens and three-dimensional look.
- The **call** command enables you to call application functions directly from the debugger.
- The **source** command enables you to specify the new name of a source file.
- The **symbol**, **shell**, and **window** commands have been added.
- Calls, Modules, and Memory windows have been added.
- Enhanced AREXX support is provided.
- Cross-referenced online help is provided in a separate window.
- The **log** command provides a Dialog window command logging capability.
- The **jump** command provides the ability to change the current location.

Terminology

This book frequently uses terms that you may not be familiar with. The following definitions are new or have unique meanings to the CodeProbe documentation:

breakpoint

is an address used to control program execution. Program execution is suspended when the breakpoint address is reached.

call frame

is the stack area associated with a function's formal parameters, automatic variables, temporaries, saved registers, and the return address. There is one call frame per function invocation.

cross-debugging

is a debugging technique that uses two machines that are linked together. CodeProbe runs on one machine, and the application runs on the other. (See host machine and target machine later in this list.)

environment

in CodeProbe documentation, *environment* means the state of the machine. The state of the machine consists of the contents of memory, the contents of the registers, the contents of the stack, and the point of execution in the source code.

The *run environment* is the actual environment at which you are stopped.

The *user environment* is the last environment set with the `env` command. This environment is used when displaying registers and variables. It is reset to the run environment after a `go` command or any of the commands that step the program.

host machine

when cross-debugging, the *host machine* is the machine that is used to display the CodeProbe user interface. The actual debugging session takes place on this machine. (See cross-debugging earlier in this list.)

image

is an executable program, an AmigaDOS shared library, or an AmigaDOS device. The name of the image is the same as the name of the disk file from which the image is loaded. An image can be specified with several CodeProbe commands as part of the location parameter.

line boundary

is the imaginary boundary in the machine code that specifies the end of the code generated for a C source line.

log file

is a file created by CodeProbe that contains details of all of the activity that occurred in the Dialog window. A log session can be turned off and on with the `log` command. By default, the log file is turned off.

module

is all code generated by a single C source file. The name of the module is the same as the name of the C source file that generated it.

pass count

is the number of times that a line has been executed. The **after** parameter is used with commands such as **go** and **break** to specify a pass count.

symbol

is a series of letters and numbers that specify a function, memory location, or a variable.

target machine

when cross-debugging, the *target machine* is the machine used to run the application. (See cross-debugging earlier in this list.)

How CodeProbe Works

CodeProbe requires debugging information to be present in the executable files before you can use most of its powerful debugging features. This means that if you want to run CodeProbe and view the C source file while it is running, you must compile that C source file with one of the debugging options, such as **line**, **full**, or **symbol**. The compiler will then include information in the executable files that will enable CodeProbe to reflect accurately the state of your application program in the Source window.

The compiler supports several debugging options specifying the amount of debugging information that is passed on to the linker. The linker (**slink**) also allows you to add symbolic debugging information to modules compiled without debugging information or to strip all debugging information from the executable file. Refer to Chapter 4, “Setting Up the Debugging Environment,” for a complete discussion of debugging options.

When CodeProbe runs your program, it uses debugging information present in executable files to keep track of which C source lines correspond to which assembler instructions.

When you specify one of the **debug** options to the compiler, the compiler generates debugging information in the form of **H_DEBUG** hunks that get placed into the object module for that source. The compiler can generate two different types of **H_DEBUG** hunks, line and symbolic. It also can vary the amount of information that goes into the symbolic **H_DEBUG** hunks based on the **debug** option. Hunks and **H_DEBUG** hunks are explained in Chapter 12, “How the Compiler Works,” in the *SAS/C Development System User’s Guide, Volume I: Introduction, Compiler, Editor*.

If you specify **debug=line**, then the compiler generates an **H_DEBUG** hunk that contains the source filename and a table of source line numbers to address offset pairs. CodeProbe uses this table to match up the line numbers for the source with the assembly instructions as your program is executing.

In addition to **debug=line**, if you specify the **addsym** option to the linker (**slink**), then **slink** generates an **H_SYMBOL** hunk in the final executable file. The **H_SYMBOL** hunk contains a table of global symbols and their offsets into the data section. If you compile and link using the **sc** command, the **addsym** option is added automatically to the **slink** command line, provided that you specify any debug option except **nodebug**.

Using the global symbol information, CodeProbe can locate the start of the symbol but not its size or type. CodeProbe assumes all such symbols are of type **long**. This feature can be useful if you are debugging large projects on small machines where there might not be enough memory to load symbolic debugging information.

If you specify the **debug=symbol** or **debug=full** options to the compiler, then the compiler generates an **H_DEBUG** hunk containing high-level symbolic debugging information for the source. This includes information for external identifiers, information for structure or union tags and typedefs, local variables, and so on. CodeProbe searches this information when trying to display an item in C source mode.

Controlling CodeProbe

Among CodeProbe's controls are pull-down menus, Intuition-based windows, and mouse support. The debugger also has powerful commands that can be entered in the Dialog window. In addition, CodeProbe offers a number of special functions and keyboard shortcuts, including programmed function and menu accelerator keys to maximize efficiency.

Debugging a Program

Debugging a program using CodeProbe involves three basic steps:

1. Compile with debug information. (Use one of the **debug=** options with the **sc** command.)
2. Invoke the debugger.
3. Execute the program under debugger control.

The next two sections of this chapter present the basic information you need to perform these steps.

Compiling and Linking Your Program

Before invoking CodeProbe you must compile and link your program as described in Chapter 8, “Compiling and Linking Your Program,” of the *SAS/C Development System User’s Guide, Volume I: Introduction, Compiler, Editor*.

The degree to which the debugger can access symbols defined in any module is determined by the compiler and linker options. Refer to Chapter 4, “Setting Up the Debugging Environment,” to determine which options you need to use. In most cases, the **debug=symbolflush** option to the **sc** command specifies sufficient information to debug your program. A typical way to compile and link your program is with the following commands:

```
sc debug=symbolflush link filename.c
```

The **sc** command invokes the compiler, and **debug=symbolflush** (or **debug=sf**) causes the compiler to generate line number information, generate symbolic debugging information for variables in that module, and make sure those variables are flushed back to memory at line boundaries. The **link** option causes **sc** to invoke the linker. The **addsym** option is automatically passed to the linker because of the inclusion of a debug option. The **addsym** option forces **slink** to generate symbolic information for global symbols in libraries and for any file that does not have debug information. The **filename.c** argument specifies the source file.

Running CodeProbe

This section describes how to

- ☐ use the **cpr** command to invoke CodeProbe from the Shell
- ☐ use the **quit** command to stop the debugger at any time
- ☐ use CodeProbe to run applications from either a Shell or Workbench screen
- ☐ run CodeProbe as a line-mode debugger.

The cpr Command

The **cpr** command is invoked from a Shell to start CodeProbe for the first time during your debugging session. The **cpr** command uses the following syntax:

```
cpr [cpr-options] application-name [application-arguments]
```

The *application-name* and *application-arguments* parameters are determined by your application. You can specify any of the following *cpr-options*:

-buffer	-noprofile	-wdialog
-cli	-screen	-wregister
-command	-startup	-wsource
-i	-w	-wwatch
-line	-wb	-cli

Each of these options is discussed in the section “Choosing Command-line Options” in Chapter 4.

When the **cpr** command is invoked, CodeProbe makes the following assumptions:

- The program specified on the command line exists in the current directory or somewhere on the command search path, as specified by the AmigaDOS **path** command.
- The modules making up the program to be debugged have been compiled with one or more of the debugging options discussed earlier and have been linked using the **addsym** option of the linker.
- Source files for modules that you will be accessing exist in either the current directory, your special source path list (as specified by the **opt search** command), or the directory in which the module was compiled.

The quit Command

The **quit** command can be issued from the Dialog window to stop the debugger at any time. However, you should allow your program to run to completion before using the **quit** command. While CodeProbe will attempt to do some cleanup for your program by calling the normal C **exit** routine, you should allow your program to do its own cleanup. Refer to Chapter 9 for more information about the **quit** command.

Workbench Support

CodeProbe can be run from the Amiga system’s Workbench screen as well as from a Shell. Furthermore, CodeProbe itself can run applications in either a Shell or Workbench environment from either location. By default, CodeProbe runs its application in the same type of environment that invoked the debugger.

The following options to the **cpr** command are used to change the default environment:

- cli** forces the application to run under the debugger as if it were invoked from a Shell.
- wb** forces the application to run under the debugger as if it were invoked from the Workbench screen.

To facilitate debugging from the Workbench screen, a small stub tool called **DEBUG** can be placed in a project directory along with a corresponding tool icon. Both the tool and its icon may be found in the **sc:starter_project** drawer. **DEBUG** invokes CodeProbe and passes to it all Workbench startup messages. The **SCSETUP** utility automatically copies **DEBUG** and its icon into your project directories.

To invoke CodeProbe from the Workbench screen, click once on the **cpr** or **debug** icon. Then, while holding down the Shift key, double-click on the icon for the application's executable.

To set up CodeProbe command-line options, use the Workbench screen's Information menu item on the **cpr** or **debug** icon from the Workbench menu. Place the debugger command-line options in the ToolTypes field of the icon. The debugger will parse the options as if they were entered on the command line. The options may be placed on one ToolType edit line or they can be spread across several lines.

To invoke CodeProbe on a program requiring command-line options, edit the **debug** icon the same way that you would edit it to set up CodeProbe command-line options. Add the **-cli** option as the last **cpr** option and follow it with the name of the executable file followed by its command-line options.

By default **debug** invokes **sc:c/cpr**. If you rename or move **cpr**, **DEBUG** can find it if you assign the full pathname of the debugger to the environment variable **sc/cprpath**. This is done easily using the AmigaDOS **setenv** command as follows:

```
setenv sc/cprpath pathname
```

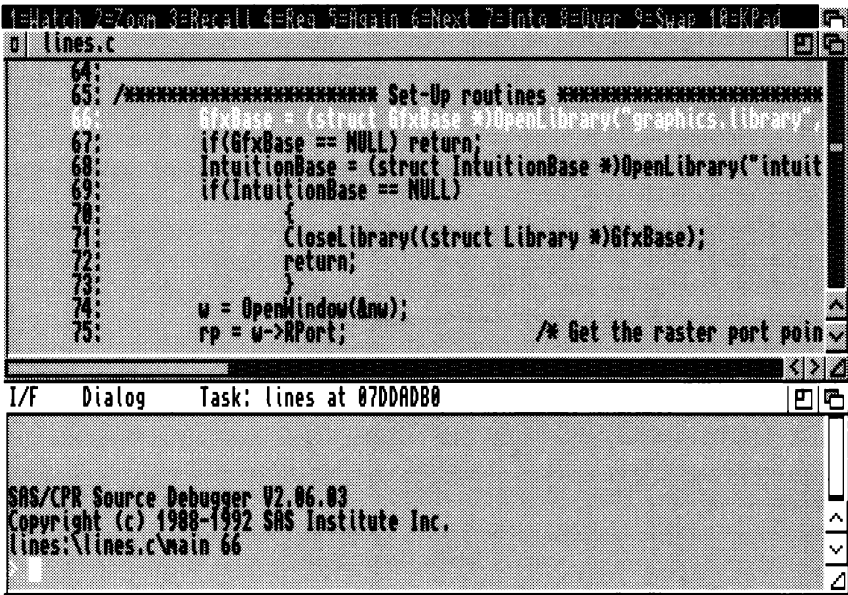
Line Mode You also can run CodeProbe as a line-mode debugger by starting it with the **-line** option. In line mode the debugger displays its output in the current Shell window. The **-line** option can be used to start the debugger from a remote Shell that has been started from an **rlogin** or from a Shell started across a serial line. For information about starting a Shell over a serial line, see the description of the AUX: device in *The AmigaDOS Manual, 3rd Edition*.

CodeProbe Windowing Interface

CodeProbe provides a sophisticated windowing interface with multiple windows and pull-down menus. Windows are used to issue debugger commands, to receive output from CodeProbe commands, or to browse through the source code for your program.

Screen Layout CodeProbe starts with the Source and Dialog windows open, as shown in Display 1.1.

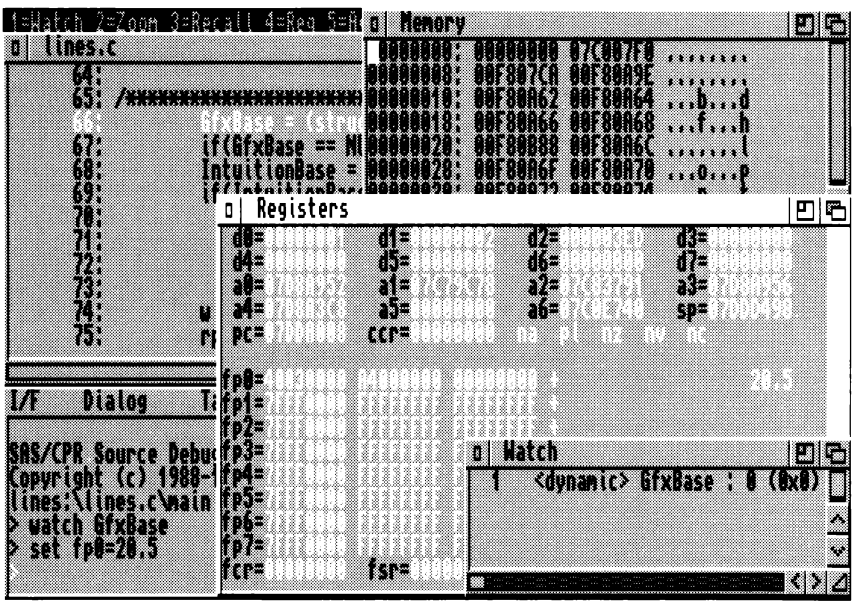
Display 1.1
Initial CodeProbe
Screen



Once CodeProbe is running, you can open, close, move, and resize windows. Display 1.2 is a sample CodeProbe screen showing the following windows:

- ☐ Source window
- ☐ Dialog window
- ☐ Memory window
- ☐ Register window
- ☐ Watch window.

Display 1.2
CodeProbe Windows



CodeProbe Windows

CodeProbe has 11 types of windows. You control the actual number of windows open during a debugging session. No matter how many windows are displayed, one of them will be the active window that is currently receiving, or waiting for, input. The title bar of the active window is highlighted (AmigaDOS Version 2.0 changes the color, and Version 1.3 ghosts the title bar when a window is not activated). The following list explains the function of each type of window:

Calls window

displays a stack traceback showing the calling sequence for the program being debugged. Each level of the traceback represents the state of the machine when a function was called. The state of the machine is also known as its environment. By double-clicking on a desired traceback entry in the Calls window, you can return CodeProbe to a previous environment. You will then see the code that was being executed at that time in the Source window, and you can see the contents of the registers as they were before the call in the Register window. The state of memory cannot be replicated for previous environments, so the Memory window reflects your current environment. A message is also displayed in the Dialog window informing you of your current location.

As an example, suppose you have a `main` function that calls a function named `sort`, which in turn calls a function named `swap`. When the execution of your program reaches the `swap` function, the calling sequence `main`, `sort`, `swap` is displayed in the Calls window. By double-clicking on the entry for the `main` function, you can return the machine to the state it was in when `main` called `sort`. You will see the source code that was executed and be able to view what was in the registers at the point at which `main` called `sort`; however, your program will still be executing at the same point in the `swap` function. Therefore, you can examine local variables, and they will have the values they had when `main` called `sort`, but global variables may have changed. If you single-step through the program, the code displayed in the Source window returns back to the last line executed in `swap`, and execution continues.

Dialog window

enables you to enter commands and review the results of those commands. Many of the same commands available through the pull-down menus can be invoked directly at the command line in the Dialog window.

Help window

displays help information for each of the CodeProbe commands as well as other selected topics.

Input window

automatically opens to request input for a task you are performing. Input windows allow you to enter data, such as selecting a value for the default number of lines to be displayed by the `list` command. You can exit out of an Input window without entering input by pressing the escape (Esc) key.

Memory window

displays a dump of memory in both character and hexadecimal format or disassembles code.

Message window

displays text passed to it by the `wmsg` command.

Modules window

displays a list of all the modules within the application program.

Output window

displays output that is directed to the terminal by your C program from functions such as `printf`. The Output window is the initial Shell if the program is run from the Shell, or the Shell Workbench window opened by the startup code otherwise.

Register window

displays the contents of the registers. This window is most useful to assembly-level programmers. It is also valuable for checking registerized variables defined in C source code. Once opened, it displays the current contents of all registers and the current values of the processor flags. Any registers that have been modified since the last breakpoint are highlighted. The Register window can be opened by pressing the F4 key.

Source window

displays the source code for the module currently being executed. As execution proceeds, the Source window is continually updated to reflect the current position. The Source window highlights the current position (the line about to be executed) and any breakpoints.

Watch window

watches and displays the value of variables. For instance, you can set a watch break such that the application will be stopped as soon as the watch variable changes value. The Watch window can be opened by pressing the F1 key.

Opening Windows

With the exceptions of the Input and Message windows, you can use the View menu to open any window. (See “Pull-Down Menus” later in

this chapter.) Input windows open automatically when user input is required. The Message window is used to display messages from scripts and is opened automatically to display output from the `wmsg` command. The Help window can be opened with the Help icon or the `help` command, as well as from the menu. You can also use the `window` command to open any of the windows.

Entering Commands

You can issue debugger commands in several different ways. The methods of issuing commands are presented in the following order:

- entering commands from the Dialog window
- using pull-down menus to enter commands
- using special function keys to enter commands
- using menu accelerator keys to enter commands.

Command Line The most flexible method of entering debugger commands is by typing them in at the command line of the Dialog window. The command line is marked by the `>` prompt. You can enter any of the debugger commands described in Chapter 9 in this manner.

Command Syntax

In order to understand the individual command descriptions, you must first understand the syntax conventions described in the “Using This Book” section.

Command Name Shortcuts

Command names need not be entered in their entirety. Generally, only the minimum number of characters necessary to uniquely identify a command is required.

The Help menu, which is displayed in the Dialog window when you enter the `help` command, shows required keys in caps for command names while the rest are shown in lowercase. For example, `RETurn` in the Help menu indicates that only the first three letters of the command are required, but typing in `return` or `retur` would work also.

The syntax conventions used in this book show the optional portion of a command in square brackets [].

Command-line Editing

CodeProbe commands can be typed on the command line of the Dialog window in either uppercase or lowercase. The following command-line editing techniques are useful when entering commands in this manner.

Determining Editing Modes

The editing mode affects the way characters are typed on the command line as well as the functioning of the numeric keypad. The current editing mode can be determined by examining the left end of the Dialog window's title bar.

The first character in the title bar will be an **I** if in insert mode or **O** if in overwrite mode. When you are in insert mode, any characters you type on the command line are inserted between existing characters. When you are in overwrite mode, any characters you type overwrite any existing characters. The **Ins** key on the numeric keypad is used to switch between insert and overwrite mode. (Use the zero key to switch between insert and overwrite modes on the A1000.)

The second character in the title bar is always a slash (/), and the third character will be an **F** if the numeric keypad is in function mode or an **N** if in numeric mode. The keypad must be in function mode to use the editing functions described in the next section. To toggle between the numeric and function modes, press **F10**.

Editing Function Keys

Three control key sequences in function mode and the numeric keypad are used to perform command-line editing functions. In order to perform these functions you must be editing in function mode. Table 1.1 describes the command-line editing functions.

Table 1.1
*Command Line
Editing Functions*

Keys	Description
Ctrl X	erases the current line
Ctrl A	toggles insert/overstrike
Ctrl keypad-1	erases from the cursor to the end of the line
keypad-1	moves the cursor to the end of the line
keypad-7	moves the cursor to the start of the line
keypad-0	starts inserting at the cursor
keypad-. (Del)	deletes the character under the cursor
keypad-4 (←)	moves the cursor left one position
keypad-6 (→)	moves the cursor right one position
Shift keypad-4 (←)	moves the cursor left one word

(continued)

Table 1.1 (continued)

Keys	Description
Shift keypad-6 (→)	moves the cursor right one word
keypad-Enter	sends the line to the program as input
keypad-8 (↑)	recalls the previous Dialog window command
keypad-2 (↓)	goes to the next Dialog window command
keypad-9 (PgUp)	moves the page up
keypad-3 (PgDn)	moves the page down

Extending Commands across Lines

A command line can extend across several input lines by ending all but the last line with a backslash (\) followed immediately by a carriage return. A command line can be up to 256 characters in length.

Using Escape Characters

There are three places where you might want to use escape characters:

- in character strings
- in character constants
- in commands.

C character strings consist of text surrounded by double quotes (""). Standard C escapes starting with the backslash character are supported as shown in Table 1.2.

Table 1.2
*Escape Sequences in
Character Strings*

Escape Sequence	Meaning
\n	newline (0x0a)
\t	tab (0x09)
\b	backspace (0x08)
\r	return (0x0d)
\f	form feed (0x0c)
\v	vertical tab (0x0b)
\NNN	octal constant (NNN is 3 octal digits.)
\xNN	hex constant (NN is 2 hexadecimal digits.)

In a character string, a backslash followed by any other character will be interpreted as itself. Therefore, "\\\" is a string consisting of a single backslash, and "a\"b" is three characters long, an "a", a double quote, and a "b".

C character constants consist of a single character surrounded by single quotes (' '). The standard C escape sequences shown in Table 1.2 can be used with character constants.

Escape characters also can be used in CodeProbe commands. Table 1.3 lists escape characters that can be used outside of character strings and constants.

Table 1.3
*Escape Sequences in
Commands*

Escape Sequence	Meaning
\\	backslash
\;	command delimiter
\,	argument delimiter
\"	character string/constant delimiter
\'	constant delimiter/constant delimiter
\(	when or macro delimiters
\)	when or macro delimiters

Outside of quotes, a backslash followed by any other character is not assigned any special meaning; it is interpreted as two distinct characters. Therefore, "b \mod\func" sets a breakpoint at function **func** of module **mod**, and not at a function called **modfunc**.

Escape sequences are especially useful in the **alias** command, as in these examples:

```
alias tb where;d x      /* this is two commands */
alias tb where\;d x     /* this is one command */
```

The first example runs an **alias** command followed by a display command; the second runs a single alias command. After executing the second example, **tb** is aliased to **where;d x**.

Note, however, that these special escape sequences are not accepted in certain places that do not make sense. For example, you cannot use escape sequences in command and alias names.

Pull-Down Menus The most common debugger activities can be performed by using pull-down menus. As with other Amiga applications, you use the right mouse button to control menu functions. CodeProbe has seven pull-down menus that enable you to enter commands. Table 1.4 lists the menu items along with their command-line equivalents.

Table 1.4
*Pull-down Menu
Items*

Menu Item	Command-line Equivalent
File Menu	
Module	<code>list \module</code>
Line	<code>list [line] [address]</code>
Find Current Line	<code>env</code>
Execute	<code>execute [script-name]</code>
Refresh	<code>Ctrl L</code>
Paste	<code><none></code>
Quit	<code>quit</code>
Options Menu	
Source Mode	<code>opt source</code> <code>(c asm mixed)</code>
Radix Default	<code>opt radix (hex decimal)</code>
Context Lines	<code>opt context [integer]</code>
List Default	<code>opt list [integer]</code>
Unassemble Default	<code>opt unassemble [integer]</code>
String Length	<code>opt strlen [integer]</code>
Array Length	<code>opt arrdim [integer]</code>
Range Length	<code>opt rangelen [integer]</code>
Autoswap Mode	<code>opt autoswap (on off)</code>
Echo Mode	<code>opt echo (on off)</code>

(continued)

Table 1.4 (continued)

Menu Item	Command-line Equivalent
Instruction Bytes	<code>opt ibytes (on off)</code>
Ignore Path	<code>opt ignorepath (on off)</code>
Case Sensitivity	<code>opt case (on off)</code>
Maximum Bad Characters	<code>opt badchar [integer]</code>
Step Into ResLib	<code>opt reslib (on off)</code>
Catch New Devices	<code>opt devices (on off)</code>
Catch New Tasks	<code>opt catch (on off)</code>
View Menu	
Calls	<code>window calls</code>
Modules	<code>window modules</code>
Register	<code>window register</code>
Watch	<code>window watch</code>
Source	<code>window source</code>
Dialog	<code>window dialog</code>
Output	<code>window output</code>
Memory	<code>window memory</code>
Help	<code>window help</code>
Run Menu	
Trace Into	<code>trace</code>
Proceed Over	<code>proceed</code>
Go	<code>go</code>
Go Until	<code>go [location]</code>
Return	<code>return [value]</code>
Start	<code>start [command-line]</code>
Break Menu	
Set	<code>break [location]</code>

(continued)

Table 1.4 (continued)

Menu Item	Command-line Equivalent
List	blist
Clear	bclear [arguments]
Enable	benable [arguments]
Disable	bdisable [arguments]
All Clear	bclear *
Watch Menu	
Set	watch [location]
Break	wbreak [arguments]
List	wlist
Clear	wclear [arguments]
Enable	wenable [arguments]
Disable	wdisable [arguments]
All Clear	wclear *
Memory Menu	
Size (Long, Word, Byte)	<none>
Display As (Data, Code)	<none>
Base Address (Static, Dynamic)	<none>

The File menu has two items called Line and Find Current Line that automatically list lines of source code in the Dialog window. These options are displayed as Address and Find Current Address when the debugger is in assembler mode. When prompted for an address after choosing the Address item, you should enter a hexadecimal value. A '0x" prefix is not required. The Source window will disassemble code beginning at the specified address.

**Function and
Special Keys**

The actions of the function keys (F1 through F10) are indicated on the debugger screen title bar. In addition to the function keys, there are three keys with special functions: Help, Ctrl L, and Ctrl W. These keys

are not shown in the title bar. Table 1.5 explains the use of the function and special keys in more detail.

Table 1.5
*Function and Special
Keys*

Key	Action
F1	toggles for opening/closing the Watch window
Shift F1	toggles for opening/closing the Source window
F2	zooms or unzooms the active window to the size of the screen
F3	recalls the last command *
Shift F3	recalls the previous command *
F4	toggles for opening/closing the Register window
F5	re-executes the last command *
F6	activates the next window
F7	steps into a line/instruction (trace)
F8	steps over a line/instruction (proceed)
F9	toggles for swap screens between the debugger and application screens
F10	toggles the numeric keypad mode
Help	executes the help command
Ctrl L	refreshes the screen
Ctrl W	refreshes the screen

* The Dialog window must be active to recall or re-execute commands.

**Menu
Accelerator
Keys**

Menu accelerator keys are familiar to most Amiga system users. Menu items with shortcuts can be executed from the keyboard by striking the **right-Amiga** key in combination with another specified key. If a menu item has an accelerator key, it is displayed on the menu after the item. The actions performed by the menu accelerator keys are shown in Table 1.6.

Table 1.6
Menu Accelerator
Keys

Key	Menu	Action
A	Break	Clears all breaks
B	Break	Sets a breakpoint
C	Break	Clears a breakpoint
D	Break	Disables a breakpoint
E	Break	Enables a breakpoint
F	File	Refreshes the screen
G	Run	Executes the <code>go</code> command
H	File	Finds the current line
L	Break	Lists the breakpoints
M	File	Changes the current module
N	File	Finds the line/address
P	Run	Proceeds (steps over the current line/instruction)
Q	File	Quits the debugger
R	Run	Returns from the current function
S	Run	Starts (restarts the program with same command line)
T	Run	Traces (steps into the current line/instruction)
W	Run	Where (displays stack frame backtrace)
X	File	Executes a debugger command script from a file

2 Sample CodeProbe Session

25	<i>Introduction</i>
25	<i>A Walk through a Typical Application</i>
25	<i>Starting the Debugger</i>
26	<i>Using the go and restart Commands</i>
26	<i>Using Online Help</i>
27	<i>Controlling the Debugging Session</i>
34	<i>Quitting the Debugger</i>

Introduction

The best way to understand how to use CodeProbe is to walk through a sample debugging session. This chapter provides an example session that demonstrates several debugger commands and options. It also explains how to maneuver in the debugger's windows.

The chapters that follow this one provide the necessary details to help you get the most out of CodeProbe.

A Walk through a Typical Application

The source code for the sample program, `lines.c`, used in this chapter is located in the `sc:examples` drawer.

Starting the Debugger

To start your debugging session you must first compile and link the application program and then invoke the debugger. This chapter provides instructions for performing these tasks with the sample program, `lines.c`. Additional information about compiling, linking, and invoking the debugger is located in Chapter 1, "Introduction to CodeProbe," and Chapter 4, "Setting Up the Debugging Environment."

Compiling and Linking the Program

To compile the sample program, `lines.c`, invoke the compiler, `SC`, with the `debug=symbolflush(sf)` option. This option outputs full debugging information for those symbols and structures referenced by the program. The `link` option invokes the linker automatically.

```
sc debug=sf link lines.c
```

Invoking the Debugger

To start the debugger, type in `cpr` followed by the executable program's filename.

```
cpr lines
```

Note that the Dialog window and the Source window open by default. The Dialog window highlights the current edit line. In the same manner, the Source window highlights the next line of the program's source code to be executed.

Using the `go` and `restart` Commands

In order to use the debugger efficiently, you need to become familiar with both the `go` and the `restart` commands. The `go` command begins execution of your application program, which runs until a breakpoint is hit or the program exits. The `restart` command causes the application program to be reloaded and execute up to the first line of the `main` function. Both of these commands are described in Chapter 9, "Commands and Built-in Functions."

When you invoke CodeProbe, a `go main` command is automatically issued unless you specified the `-startup` option when you invoked the debugger. This initial `go` command causes your application to run to the first line of the `main` function. You can always return to this point by entering as many `go` commands as necessary to cause your application program to run to completion and then issuing a single `restart` command.

Using the `restart` command incorrectly can crash your machine. You can issue a `restart` command at any point in your program, which causes execution to jump back to the beginning of your code and run up to the first line of the `main` function. However, `restart` does not finish executing the rest of your program before it returns to the beginning, and cleanup procedures located at the end of a program are not executed. These cleanup procedures return memory and registers to a state expected by the operating system and other programs. Failing to complete execution of your program before restarting or exiting CodeProbe may leave the machine in an unacceptable state that will later cause a crash.

Using Online Help

At any point in your debugging session, you can use the `help` command to get online help regarding debugger commands and other CodeProbe features. The `help` command can be issued from the command line using either of the following forms:

```
h[elp] [command]
? [command ]
```

The optional *command* parameter specifies the debugger command for which help information is to be displayed. A brief description, a summary of command syntax, some sample uses, and references to other commands is provided.

If you enter the **help** command without the *command* parameter, a list of debugger commands is displayed along with other topics for which online help is available. To make a selection from the list, double-click the mouse pointer on one of these commands or topics.

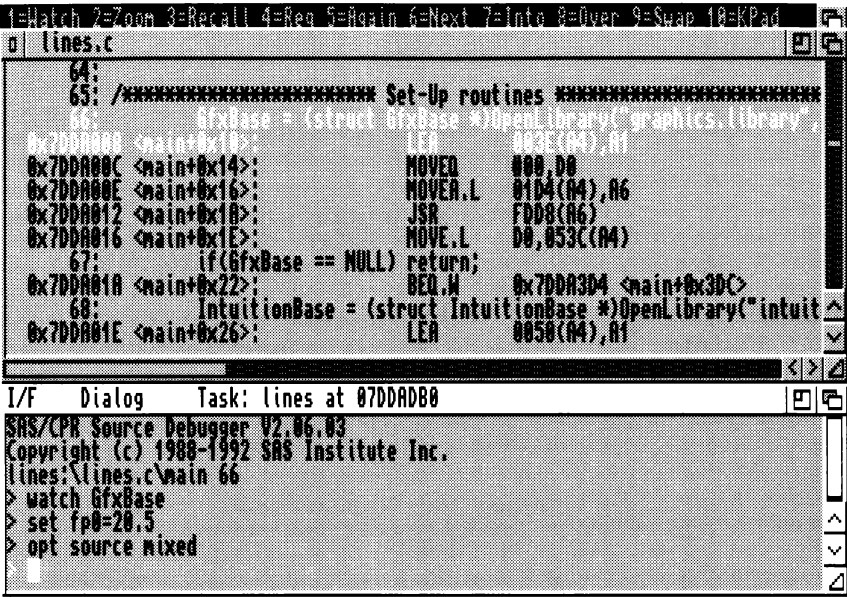
Controlling the Debugging Session

After you have invoked the debugger, you control your debugging session by entering debugger commands as described in Chapter 1, “Introduction to CodeProbe.” Reference information for each of the debugger commands is provided in Chapter 9, “Commands and Built-in Functions.”

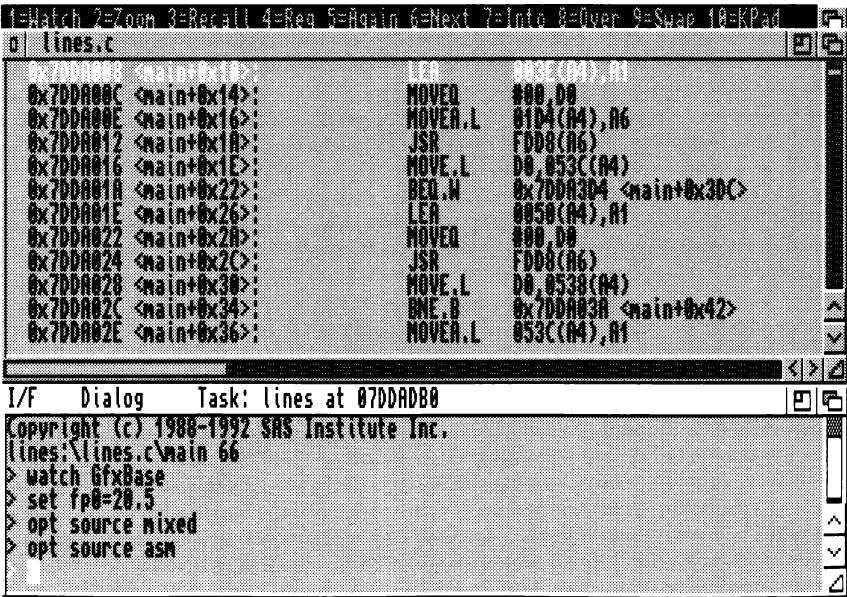
Setting the Source Mode

The *source mode* controls the way the source code is displayed in the Source window. You can use a pull-down menu to change the source mode. Hold down the right mouse button and drag the pointer through the menu items under the Options menu. The **Source Mode** option appears as a menu item and indicates **C** as the default mode. When the **Source Mode** option is highlighted, three subitems are offered as choices: **C**, **Asm**, and **Mixed**. Modifying the selected choice alters the appearance of the lines in the Source window accordingly. For instance, Display 2.1 shows the results of selecting mixed mode, while Display 2.2 shows the results of selecting assembly mode. When debugging a C application, you should usually choose C mode. Refer to the section “Source Mode” in Chapter 4 for additional information.

Display 2.1
Source Window
Showing Mixed Mode



Display 2.2
Source Window
Showing Assembly
Mode



Activating Autoswap Mode

You also can activate **Autoswap mode** from the options menu. When autoswap mode is on, the screen containing the application's output is pushed to the front each time CodeProbe gives control to the application. When a breakpoint is reached, the debugger screen is pushed again to the front. If autoswap mode is not on, you must change to the output screen manually using function key F9 or the keyboard shortcut **left-Amiga-m**. Regardless of what method you use to swap screens, you must activate each screen by first clicking the left mouse button before entering any input.

Stepping Over Code

One of most useful features of the debugger is the ability to step through code. The **proceed** command enables you to single step over function calls. If you need to step into a called function you can use the **trace** command. Since the **proceed** command is the most frequently used command, CodeProbe is designed so that pressing the Return key causes the **proceed** command to be executed. Step over the first few lines of code as follows:

```
lines:\lines.c\main 71
>
lines:\lines.c\main 72
>
lines:\lines.c\main 73
>
lines:\lines.c\main 74
```

Select the Options menu and turn **Autoswap mode** on. Now step over several more lines of code. Notice the quick flashing due to the display of the application window (the initial Workbench CLI window) during each step.

```
>
lines:\lines.c\main 79
>
lines:\lines.c\main 80
>
lines:\lines.c\main 81
>
lines:\lines.c\main 82
```

You will notice also that the debugger does not stop at every line. This is due to the fact that code is not generated for all lines. Code is not generally produced for lines that declare variables, such as lines 66-68, or lines that are blank or contain only comments.

Continuing Execution

If you do not want to take the time to execute each line individually, you can use the **go** command, which executes your application until it either encounters a breakpoint (a flag to stop execution) or the end of the program. You should always use **go** to complete execution of your program before quitting CodeProbe or using the **restart** command.

Displaying a Structure Or an Array

The **display** command, which can be abbreviated as **d**, allows you to display structures, arrays, and variables easily. For instance, to display the structure **nw**, issue the following command in the Dialog window:

```
>display nw
struct NewWindow {
    LeftEdge = 100 (0x64)
    TopEdge = 100 (0x64)
    Width = 300 (0x12C)
    Height = 100 (0x64)
    DetailPen = '\x02' 2 (0x02)
    BlockPen = '\x01' 1 (0x01)
    IDCMPFlags = 514 (0x202)
    Flags = 1039 (0x40F)
    FirstGadget = 0x0
    CheckMark = 0x0
    Title = 0x00226028
    Screen = 0x0
    BitMap = 0x0
    MinWidth = 50 (0x32)
    MinHeight = 50 (0x32)
    MaxWidth = 640 (0x280)
    MaxHeight = 400 (0x190)
    Type = 1 (0x1)
}
```

Use F2 to zoom the Dialog window to full size so that the entire structure can be viewed at once. The value of the character pointer **Title** will probably be different on your screen, but the other values

should be the same. When you have finished viewing, press F2 to restore the Dialog window to its original size.

To display the two-dimensional array `ox`, issue the following command from the Dialog window:

```
> display ox
{
[0] = {
    [0] = 0
    [1] = 0
    [2] = 0
    .
    .
    .
}
```

Then press F2 again to zoom the Dialog window to full size. While the values are not noteworthy, you can get an idea of the format that is displayed. Notice that the entire array does not fit in a full-size window. To see the beginning values, use the mouse and the Dialog window's vertical scroll bar to scroll backwards.

Individual members of structures and unions, as well as elements of arrays, can be displayed using the `display` command, as in the following examples:

```
> display nw.Type
1 (0x1)
> display x[1]
0 (0x0)
> display w->RPort
0x07F000FC
> display ox[1][5]
0 (0x0)
> display nw.Title
0x00226028
> dzero nw.Title
"Nervous Lines"
```

You probably will find different results for the third and fifth examples. Notice that the last command `dzero` indicates that `nw.Title` should be treated as a pointer to a null-terminated string and its contents displayed rather than the pointer itself.

Setting a Breakpoint

A *breakpoint* is an address at which program execution stops whenever encountered. To set a breakpoint, move the pointer to a given line of code in the Source window. Using the left mouse button, double-click on the line. This sets a breakpoint on the line, and the line is highlighted. If there is no code generated at the line, an error message is displayed in the Dialog window. The breakpoint can be removed easily by double-clicking again on the particular line of code.

Executing up to a Breakpoint

Use your mouse to scroll down to line 111 in the Source window and double-click on line 111 to set a breakpoint. Now execute the program up to the breakpoint by typing the `go` command on the command line. The breakpoint line is now highlighted in a different way to indicate that execution has proceeded to that point. Double-clicking on the line turns off the breakpoint. The line reverts back to normal, highlighting just the text, indicating this is the next line to be executed.

Displaying the Value and Type of a Variable

The debugger will display a variable's value and tell what type it is. Enter the following commands in the Dialog window:

```
> display i
2 (0x2)
> whatis i
auto register variable in d7 (2 bytes)
short i;
> whatis x
extern at 0022625E (8-byte array of 2 4-byte element(s))
int x[2];
> whatis nw
extern at 00226036 (48 bytes)
struct NewWindow nw;
```

Notice that the `whatis` command can provide information about the location and attributes of any variable. This is particularly useful for keeping track of automatic variables, which may be placed in registers by the compiler, even though they were not explicitly declared as type `register`.

Using the watch and watch break Commands

A *watch* allows you to monitor a variable or area of memory to see when the value of the variable changes or when the area of memory is

altered. Create a watch for the variable `x` by issuing the following command:

```
> watch x
```

Open the Watch window by pressing the F1 key. You may have to adjust the size of the Watch window to see the entire display. Note that the output shown in the Watch window is a hex dump of the variable `x`:

```
1 <dynamic> x : 00 00 00 51 00 00 00
```

The number 1 at the left of the display identifies the watch. It is used by other commands that manipulate the Watch window.

A *watch break* is similar to a watch but acts as a breakpoint when the variable or area of memory is changed. Create a watch break for the variable `y` by issuing the following command:

```
> wbreak y
```

At this point the Watch window should appear as follows:

```
1          <dynamic> x : 00 00 00 51 00 00 00
2! <static 0x36C472> y : 00 00 00 30 00 00 00
```

Notice that the watch break is highlighted and has an exclamation mark (!) following the watch break number. The exclamation mark indicates that it is a watch break and not simply a watch.

Executing Until the Variable Changes Value

The Watch window is updated when control is returned to you by the debugger, such as at a breakpoint, and the new values of the watched areas are displayed. Note that running with set watch breaks can be a slow process because the CPU executes the application in trace mode. This means that control returns to the debugger after every instruction. The debugger must then check the memory for all watch breaks before restarting the application. The watch break feature is a powerful tool, and it is particularly useful for locating obscure bugs that cause memory to be overwritten from unusual places in the code; however, it should not be used indiscriminantly.

To see the effect of the watch break, click in the Dialog window and issue the **go** command:

```
> go
break (watch #2)
  2! <static 0x36C472> y: 00 00 00 38 00 00 00
lines:\lines.c\main 117
```

Notice that the debugger reports the watch break number that caused the breakpoint to occur. Also, notice that the line that modified the variable was actually the previous line. Line 116 is the next line to be executed.

Clearing All Watches and Watch Breaks

To clear watches and watch breaks, issue the following Dialog window command:

```
> wclear *
```

The asterisk (*) is a wild card representing all watch and watch break identifiers.

Now that all watches and watch breaks have been cleared, issue the **go** command to start the program **lines.c**. Notice that no breakpoints can be reached now, so the program will continue until completion. Standard input/output is written to the application window on the Workbench screen.

To see the application as it is running, you can use the **left-Amiga-m** key sequence to swap screens. The demo will run until the close gadget is activated on the **lines.c** application window. After closing the application window, swap screens back to CodeProbe.

The Dialog window should now display the following:

```
Program exited with code 0.
```

The code value cited, 0 in this case, is the return code from the program.

Quitting the Debugger

To quit the debugger, issue the command **quit** at the command line, or use the pull-down menu item in the File menu.

3 Debugging Your Program

35	<i>Introduction</i>
36	<i>CodeProbe Commands</i>
37	<i>Sample Program 1</i>
37	<i>Source Code</i>
39	<i>Running Program 1</i>
39	<i>Controlling Program Execution</i>
39	<i>The go Command</i>
44	<i>Single Stepping</i>
45	<i>The trace Commands</i>
46	<i>The proceed Commands</i>
47	<i>Breakpoints</i>
48	<i>The break Command</i>
53	<i>Sample Program 2</i>
54	<i>Source Code</i>
56	<i>Running Program 2</i>
57	<i>Examining Data and Data Structures</i>
57	<i>The display Command</i>
61	<i>The dump Command</i>
66	<i>The register Command</i>
68	<i>The whatis Command</i>
69	<i>Watches and Watch Breaks</i>
70	<i>The watch and wbreak Commands</i>

Introduction

This chapter explains the use of several common CodeProbe commands to perform the following tasks:

- ☐ control program execution
- ☐ examine data and data structures
- ☐ modify data
- ☐ use watches and watch breaks.

The commands used to perform these tasks also are explained in this chapter because they can be complex to use effectively.

**CodeProbe
Commands**

This chapter introduces the following complex CodeProbe commands:

break	go	watch
display	proceed	wbreak
dump	trace	whatis

There are additional commands that are not discussed here. For a complete listing of the commands and detailed reference information, refer to Chapter 9, “Commands and Built-in Functions.”

Commands for Running Sample Programs

The following commands are useful when running the sample programs described in this chapter:

bclear
clears one or more breakpoints

blist
displays a list of all the breakpoints that have been set

restart
reloads your program and reinitializes your static data allowing you to start over from the top.

Note: The **restart** command is an advanced command. Before using it to debug your programs, you should review carefully the **restart** command in Chapter 9.

Additional Commands

In addition to the commands described in this chapter, you should become familiar with the following commands, which are described in Chapter 9, to get the most out of CodeProbe:

bdisable	rflag	wdisable
benable	set	wenable
call	symbol	wlist
jump	where	
return	wclear	

Sample Program 1

To view the commands described in the first part of this chapter, you should compile, link, and debug Sample Program 1. It is a simple program consisting of three modules (`smain.c`, `sort.c`, and `swap.c`) that first initializes an array of integers from 0 to 9 and then sorts the array into reverse order.

The examples in this chapter are designed so that you start CodeProbe only once, at the beginning of the section, and then run each example in order. Running the examples out of order can produce results that do not agree with the documentation. Extra `go` and `restart` commands are included with several of the examples to position the file back at its beginning. The same results can be achieved by entering `go`, quitting CodeProbe, and starting a new debug session with the same program. Note, however, that you should always allow your program to run to completion before restarting or quitting CodeProbe.

Source Code The following source code for Sample Program 1 is provided in the `sc:examples/debugger` drawer:

```
/* SMAIN.C */

(1)      #define ASIZE 10
(2)
(3)      void init(int *, int);
(4)      int sort(int *, int);
(5)
(6)      int array[ASIZE];
(7)
(8)      void main(void)
(9)      {
(10)         init(array,ASIZE);   /* initialize array[] */
(11)
(12)         /* Keep swapping elements until sorted */
(13)         while (sort(array,ASIZE) != 0)
(14)             ;
(15)     }

/* SORT.C */

(1)      void swap(int *, int *);
(2)
(3)      void init(int *ip, int size)
```

```

(4)      {
(5)          int i;
(6)          int *p;
(7)
(8)          p = ip;
(9)          for (i = 0; i < size; i++)
(10)             *p++ = i;
(11)      }
(12)
(13)      /* Reverse sort elements of an array */
(14)
(15)      int sort(int *ip, int size)
(16)      {
(17)          int i, s;
(18)          int *p;
(19)
(20)          p = ip;
(21)          s = 0;          /* no swaps performed yet */
(22)
(23)          for (i=0; i < size - 1; i++)
(24)              if (p[i] < p[i+1] )
(25)              {
(26)                  swap(&p[i], &p[i+1] );
(27)                  s = 1;    /* indicate a swap took place */
(28)              }
(29)
(30)          return(s);
(31)      }

```

/* SWAP.C */

```

(1)      void swap(int *x, int *y)
(2)      {
(3)          int tmp;
(4)
(5)          tmp = *x;
(6)          *x = *y;
(7)          *y = tmp;
(8)      }

```

Running Program 1

To run Sample Program 1, compile, link, and load it into the debugger as follows:

1. Compile and link Sample Program 1 using the following command:

```
sc debug=sf link sort.c smain.c swap.c
```

2. Start CodeProbe by entering the following command:

```
cpr sort
```

Controlling Program Execution

With CodeProbe you can exercise a high degree of control over program execution. You can single step through your program (at either the assembly or C source level), step over or into function calls, set breakpoints at source line or instruction addresses, or go to a specific location.

This section describes the following Dialog window commands for controlling program execution:

break

sets a breakpoint at the specified location.

go

starts the execution of the program you are debugging and continues until it hits a breakpoint or the end of the program. Execution begins at the line highlighted in the Source window.

proceed

single steps through your program or steps through it a given number of times, stepping over function calls, unless they contain a breakpoint.

trace

single steps through your program or steps through it a given number of times breaking at a function call.

The go Command

The **go** command starts the execution of the program you are debugging. This command uses the following syntax:

```
g[o] [location [after(integer)] [when(expression)]]
```

The **go** command has three different parameters: *location*, **after**(*integer*), and **when**(*expression*). Note that the parentheses around the *expression* and *integer* parameters must be typed in.

The optional *location* parameter indicates where execution should terminate. When the **go** command is used, the debugger sets a temporary (non-sticky) breakpoint at a specified location. This breakpoint is temporary in the sense that the breakpoint is deleted after it is encountered. If you already have a normal breakpoint at a particular location and execute the **go** command to that location, the original breakpoint is cleared.

The **after** clause uses the *integer* parameter to specify the number of times that the line indicated by the *location* parameter should be executed. This is called a *pass count* and program execution stops after the line is passed the specified number of times.

The **when** clause uses the *expression* parameter to specify the circumstances under which execution is suspended at *location*.

In its most general form, the **go** command can be read as: **go** until *location* has been executed *integer* number of times and stop the next time *location* is reached when the *expression* is true.

When used without parameters, the **go** command starts the execution of your program, which runs to normal completion if no active breakpoints are encountered.

Specifying a Location

The *location* parameter specifies the point in your program at which you want execution to stop. You can specify either an address or line number in a source file and a breakpoint will be set at that location. A *location* can take any of the following forms:

```

address
line
function
function line
\module\function [line]
image: function
image: function [line]
image:\module\function [line]

```

The *address* form of the *location* parameter is used to specify an explicit address constant. This form is useful for setting breakpoints at specific addresses when debugging in assembly mode.

The *line* parameter is usually a number referring to the line number where you want execution to stop. Other possibilities for *line* are the following:

- \$**
refers to the current line
- e[ntry]**
refers to the first instruction of the specified function, or the current function if none is specified
- r[eturn]**
refers to the last instruction of the specified function, or the current function if none is specified.

A line number is always relative to the beginning of the module. The *line* parameter can be used in conjunction with the *function* parameter to describe a *location*. For instance, in the form

function line

line refers to the line number of the module containing *function*. If the specified line is not inside the specified function, an error is printed. If a function is specified without a *line* parameter, the entry point to the function is assumed.

When the specified location is reached, a message is displayed describing the source filename, the function name, and the line number. If mixed or assembly mode has been specified, the information is preceded by the breakpoint address. For example, using Sample Program 1 shown earlier in this chapter, the session log displayed in the Dialog window would look as follows, if C mode is currently chosen:

```
> go swap 5
sort:\swap.c\swap 5
> opt source mixed
> go swap 7
at 0x38974C sort:\swap.c\swap 7
> opt source c
> go
program exited with code 0.
> restart
starting "sort"
sort:\smain.c\main 10
```

Here execution has stopped after the second `go` command was issued, at line 7 of the file `swap.c`, and within the function `swap`.

If the *module* is not specified, a search will be made among your modules, beginning with the current module, until *function* is found. In the unlikely event that you have defined two functions in separate modules with the same name and belonging to the same static storage class, then it is necessary to specify the *module*.

If a *function* is not specified, the currently executing function is assumed.

If an *image* parameter is specified, it refers to the name of a program or shared library that may or may not have been loaded. If it is not yet loaded, the debugger sets a deferred breakpoint in the library or program. When the library or program is loaded a breakpoint is set and program execution proceeds. If the debugger cannot find the debugging information for the image, it will halt when the image is loaded. At this point you can use the `symload` command to specify the location of the version to be debugged. For example, the following `go` command causes the program to execute until `mylib.library` is loaded, a breakpoint is then set at `LIBmyfunc`, and program execution continues. This example command will not work with Sample Program 1.

```
go mylib.library:LIBmyfunc
```

Specifying the after Clause

As previously described, the `after(integer)` parameter is used to specify a pass count, which is the number of times that the line indicated by the *location* parameter should be executed. For example, if you enter the following command in the Dialog window after starting CodeProbe with Sample Program 1, execution will stop after the `while` statement on line 13 of the program has executed 5 times:

```
> go 13 after(5)
sort:\Work:examples\smain.c\main 13
```

Specifying the when Clause

The `when(expression)` parameter is used to further control program execution. An *expression* can be any valid C expression. For example, any of the following expressions can be used with a `when` clause:

```
i == 7
i < 12 || i > 5
i != 4 && strlen(string) != 7
```

Consider the following loop in the `sort` routine of Sample Program 1:

```
for (i = 0; i < size - 1; i++)
    if (p[i] < p[i + 1])
```

You may want to stop at a point within the loop, but only after a certain number of iterations of the loop have been executed. Assuming that the `for` statement is on line 23, this can be accomplished with the following `go` command:

```
> go sort
sort:\Work:examples\sort.c\20
> go 4 when (i == 7)
sort:\Work:examples\sort.c\sort 4
```

This has the effect of planting a non-sticky breakpoint at line 24 of the current module. That is, the breakpoint will be triggered only if the `when` condition is true; otherwise execution will continue past the specified line. Note that the specified line is the line after the loop. This is because breakpoints are set at the beginning of a line, before the code on the line is executed. If the breakpoint had been set at the line on which the `for` statement occurs, the breakpoint would never have been encountered while the loop was executing. For example, if you give the following command the breakpoint would not be triggered, because `i` will never be less than 0 and the program will run to completion:

```
> go 24 when (i == -1)
program exited with code 0.
```

Breakpoints are discussed in greater detail later in this section.

If you make any mistakes in syntax when entering the condition, errors are not detected until the first time the specified location is reached and the debugger attempts to evaluate the condition. If the debugger cannot evaluate the condition, the breakpoint is triggered and an error message is displayed. The message indicates the position in the condition where syntax rules have been violated. For example, the

following message indicates that `!<` violates rules governing permitted relational symbols:

```
> restart
> go sort
> go 24 when (i !< 1)
Processing the when clause:
(i !< 1)
    ^-----syntax error
sort:\sort.c\sort 24
```

Examples

The following are examples of ways to use the `go` command:

```
go main
    goes until the entry point of function main

go return
    goes until the end of the current function

go 0xC80200
    goes until the address 0xC80200 is reached

go sort 24 when (i==5)
    goes until line 24 of the sort function when i equals 5

go init 10 after(5) when (i>4)
    goes until pass 5 at line 10 in the init function when i is greater
    than 4
```

Single Stepping The `proceed` and `trace` commands can be used to single step your program. Each command has two variations: one to single step at the assembler level and the other to single step at the C source level. The difference between the two commands lies in their handling of function calls.

The `proceed` commands step over function calls; if a function call is encountered during the execution of a `proceed` command, the function is executed without a break occurring (unless one has been deliberately set in the function), and then the next line after the function call is highlighted. Execution continues in the calling function. This is in contrast to a `trace` command, which steps into function calls, highlighting the first line of the function as the next line to be executed.

The trace Commands

The **trace** commands, **t** and **ts**, are used to single step your program. The **t** command is for single stepping assembler instructions. The **ts** command stands for trace source and is used for single stepping C code. The syntax for the trace commands is as follows:

```
t[race] [integer]
ts [integer]
```

The optional *integer* parameter indicates the number of steps to be executed; if absent, a single step is executed. A step is considered to be either one assembler instruction or a single line of C source code.

Either **trace** command will step into any function calls encountered during the step. That is, the called function is entered and execution is suspended at its first instruction.

The actions of these commands depend on the current source mode of the debugger. In C source mode, the two **trace** commands act identically, causing the execution of one line of C source code up to any imbedded function call. In assembly or mixed mode, **t** executes one machine instruction; **ts** always executes a single C source line.

Both **trace** commands step into function calls. Regardless of the mode, if the current line contains a function call, then the first line of that function is executed and subsequent **t** commands execute lines within that function. In assembly mode, if the current line contains a JSR (Jump to SubRoutine) or a BSR (Branch to SubRoutine) instruction, then that instruction is executed and subsequent **t** commands execute instructions in the called function.

Using Sample Program 1 given earlier in this chapter and starting over from **main** with the **restart** command, trace for several steps in source mode as shown here:

```
> go
program exited with code 0.
> restart
sort:\smain.c\main 10
> trace
sort:\sort.c\init 4
> trace
sort:\sort.c\init 8
> trace 3
sort:\sort.c\init 9
sort:\sort.c\init 9 (+0x2)
sort:\sort.c\init 10
```

```

> t
sort:\sort.c\init 9 (+0x2)
> t
sort:\sort.c\init 10
> t
sort:\sort.c\init 9 (+0x2)

```

The first **trace** command carries us to the **init** call in **main**. The second then moves us into the **init** function. The third illustrates the use of a numeric argument to execute several steps.

To step by a single machine instruction, you can set the source to either **asm** or **mixed**. For instance, you can set your source mode to **asm** by using the pull-down Options menu or by issuing the following command:

```
opt source asm
```

Similarly, you can set your source to **mixed** with the following command:

```
opt source mixed
```

At this point, you can enter the **trace** command to step over an assembler instruction.

If source mode is set to **asm** and no line information is available, using the **ts** command is regarded as an error, and an appropriate error message is displayed. In mixed mode, the **trace** command causes stepping by machine instruction while the **ts** command causes stepping by C source line.

The proceed Commands

The **proceed** commands, **p** and **ps**, also are used to single step your program or to step it a given number of times. Unlike **trace**, however, a **proceed** command steps over function calls rather than stepping into them. That is, if a call to a function is encountered in the process of the step, the function is not stepped into as in the **trace** commands.

The syntax for the **proceed** commands is as follows:

```

p[roceed] [integer]
ps [integer]

```

The *integer* parameter indicates the number of steps to be executed. The **proceed** and **ps** commands are analogous to the **trace** and **ts** commands with respect to source mode.

Looking at the previous example, notice how the results of the **proceed** commands differ compared to the **trace** commands (C source mode is used again):

```
> go
program exited with code 0.
> restart
sort:\smain.c\main 10
> proceed
sort:\smain.c\main 13
> proceed
sort:\smain.c\main 14
> proceed
sort:\smain.c\main 13
```

Here, a single step has been made for each source line of the function **main**. Function calls were made in the process of this stepping, but none of the called functions were entered.

Pressing the Return key is equivalent to entering the **p** command. Continuing where the previous example left off, the following example demonstrates how this works:

```
> <Return>
sort:\smain.c\main 14
> <Return>
sort:\smain.c\main 13
```

Using the **ps** command in assembly mode is regarded as an error if no source line information is available, while **proceed** can be used in mixed or assembly mode to single step by machine instruction. The **ps** command can be used in mixed mode to step by C source line in a manner similar to the **ts** command.

Breakpoints A breakpoint is an address at which program execution stops. The **break** command enables you to suspend the execution of your program or to execute a set of debugger commands at any point you choose.

By setting conditional breakpoints or by using a pass count to specify the number of times a program should pass a breakpoint before execution is stopped, you can suspend program execution at just the

right stage of your program. This means, for example, that you can set a breakpoint at a line within a loop so that the program stops after going through 500 iterations of the loop. By triggering a breakpoint in this way, you are able to easily reach the point in the program that you want to examine.

Besides breakpoint setting, additional commands allow you to manage your breakpoint list. **bclear** provides you with the ability to clear a breakpoint, **bdisable** temporarily disables a breakpoint, **benable** enables it again, and **blist** lists the current breakpoints.

The break Command

The **break** command enables you to specify a breakpoint. Breakpoints are used to suspend the execution of a program at any point or execute a set of debugger commands instead of suspending execution. You can also set and clear breakpoints by double-clicking the mouse on a line in C source mode or an assembly instruction in **asm** or **mixed** mode.

The syntax of the **break** command has two forms:

```
b[reak] $
```

```
b[reak]location [after(integer)][when(expression)]  

[tr[ace]] [q[uiet]] [te[mp]] [{cmd-list}]
```

In the first form, a breakpoint is set at the current location (which in C source mode is the current line). This is handy when you have stepped to a certain point in your program and want to set a breakpoint quickly at the current line. Here is an example:

```
> go sort  

sort:\sort.c\sort 20  

> <Return>  

sort:\sort.c\sort 21  

> <Return>  

sort:\sort.c\sort 23  

> break $  

> go  

sort:\sort.c\sort 23
```

In this case we have single stepped twice by pressing Return (the same as a **proceed** command). When the breakpoint is set with the **break** command, line 23 in the Source window is highlighted. A breakpoint also can be easily and quickly set or removed by double-clicking on the line in the Source window where you want the breakpoint.

The second form of the **break** command allows you to set a breakpoint at a specified line number and to ensure that the breakpoint is not triggered until that line has been executed a specified number of times.

For example, suppose you want to set a breakpoint at the **if** statement in the **sort** routine, but you want the breakpoint triggered only after the fifth time that statement is executed. You can use the following commands:

```
> break sort 24 after(5)
> blist
1 0xC32DCA sort:\sort.c\sort 23 (1 hit)
2 0xC32DD4 sort:\sort.c\sort 24
   after(5)
> go
sort:\sort.c\sort 24
> display i
4 (0x4)
```

The fact that the value of **i** is 4 indicates that the loop has already executed five times. You will find the pass count feature to be especially useful in debugging programs containing loops that iterate many times.

Conditional Breakpoints

Although breakpoints with pass counts are a kind of conditional breakpoint, the **break** command also allows for a more generally useful kind of conditional breakpoint. Using the **when** option, you can specify a condition that depends upon the values of variables in your program. You might, for example, want to stop at the **if** statement in **sort** when the value of **p[i]** is 8:

```
> bclear *
> break sort 26 when (p[i] == 8)
> go
sort:\sort.c\sort 26
> display p[i]
8 (0x8)
```

Any valid C expression can be used as an argument in a **when** clause.

Of course the **after** option can be used to specify a pass count in conjunction with the **when** clause. Both the pass count and the **when**

clause conditions must be met. You may, for example, want to see if there is an instance where the array element is equal to the array index after three or more executions of a given line:

```
> bclear *
> go
program exited with return code 0.
> restart
> break sort 26 after (3) when(p[i]==i)
> go
sort:\sort.c\sort 26
> dump i decimal
reg d6:          2
> dump p[i] decimal
00C28708        2
```

Attaching Actions to Breakpoints

Instead of just stopping at a breakpoint, you may want to have the debugger perform one or more actions. You can do this with the `cmd_list` parameter. The `cmd_list` parameter is a sequence of debugger commands enclosed in braces (`{ }`) and separated by semicolons (`;`). The following example shows how to use the `cmd_list` parameter to stop at line 25 of the `sort` function and have the debugger automatically display the value of `p[i]` and the value of `i`.

```
> bclear *
> break sort 26 {display "p[i] = ", p[i] ; display "i = ", i}
> go
sort:\sort.c\sort 26
p [i] = 2 (0x2)
i = 3 (0x3)
> go
sort:\sort.c\sort 26
p [i] = 2 (0x2)
i = 4 (0x4)
> go
sort:\sort.c\sort 26
p [i] = 2 (0x2)
i = 5 (0x5)
```

In fact, you can arrange for the debugger not to stop at the breakpoint but to continue by issuing the **go** command as the last command in your list, as in the following example:

```
> bclear *
> go
program exited with code 0.
> restart
sort:\smain.c\main 10
> break sort 26 when (i < 3) {dump p L 4; go}
> break sort 24 when (i >= 3)
> blist
10 0xC32D4C sort:\sort.c\sort 26
    when (i < 3) {dump p L 4; go}
11 0xC32D2C sort:\sort.c\sort 24
    when (i >= 3)
> go
sort:\sort.c\sort 26
00C31260: 00000000 00000001 00000002 00000003
sort:\sort.c\sort 26
00C31260: 00000001 00000000 00000002 00000003
sort:\sort.c\sort 26
00C31260: 00000001 00000002 00000000 00000003
sort:\sort.c\sort 24
```

In this way, you can actually see the sort taking place and you do not need to type in the **go** command repeatedly. Using the **trace** option on the **break** command eliminates the need to include a final **go** in the command list parameter. This is demonstrated in the following example:

```
> bclear *
> break sort 26 when (i > 5) {dump p L 4;
    break sort 26 when (i == 8) trace; blist}
> blist
12 0xC32D4C sort:\sort.c\sort 26
    when (i > 5) {dump p L 4;
    break sort 26 when (i == 8) trace; blist}
> go
sort:\sort.c\sort 26
00C31260: 00000001 00000002 00000003 00000004
15 0xC32D4C sort:\sort.c\sort 26
    when (i == 8) trace
sort:\sort.c\sort 26
```


Here a breakpoint is set at line 26 of `sort` and is to be triggered when `i` has a value greater than 5. When triggered, the breakpoint action displays the first four elements (16 bytes) of the array referenced by `p`, sets a new breakpoint at the same line, lists the new breakpoint, and continues execution. The new breakpoint is then triggered when its `when` condition is satisfied.

This nesting of breakpoints within action lists is a powerful feature of the debugger. Since nested breakpoints may become complex, you should take great care in using them or you may lose your place in the program when some breakpoints replace themselves with others.

You must also be aware that a breakpoint with an action may have an effect (and perhaps an unexpected one) on program execution. To consider a simple case, suppose you set a breakpoint on line 26 of `sort` and attach to it an action that will alter the value of `i`:

```
> bclear *
> break sort 26 when (i == 5) {set i = 2}
> go
sort:\sort.c\sort 26
> dump i decimal
reg d6:          2
> go
sort:\sort.c\sort 26
> dump i decimal
reg d6:          2
```

In this simple case you have managed to create an infinite loop by means of your breakpoint and action list. Such a simple mistake is easy to find, but if you have planted a number of breakpoints or if your breakpoints are deeply nested, the error might be very difficult to detect. This example also demonstrates that the action for a breakpoint is executed before control is returned to you; by the time you examine the value of `i`, it has already been set to its new value by the `set` command.

You can also use the `trace`, `quiet`, and `temp` options with the `break` command. The `trace` option causes execution to continue after the breakpoint is hit, the `quiet` option suppresses the default message that is printed when the breakpoint is hit, and the `temp`

option deletes the breakpoint after it is hit. The following example illustrates the use of these options:

```
> go
> restart
> bclear *
> break sort 26 trace quiet {display i}
> break sort 30 temp
> blist
  1 0xC32D4C sort:\sort.c\sort 26
    trace quiet {display i}
  2 0xC32D66 sort:\sort.c\sort 30
    temp
> go
0 (0x0)
1 (0x1)
2 (0x2)
3 (0x3)
4 (0x4)
5 (0x5)
6 (0x6)
7 (0x7)
8 (0x8)
sort:\sort.c\sort 30
> blist
  1 0xC32D4C sort:\sort.c\sort 26
    trace quiet {display i}
```

Sample Program 2

To view the commands described in the second part of this chapter, you should compile, link, and debug Sample Program 2. It is a simple program consisting of one module (`data.c`).

Sample Program 2 defines and initializes various data structures. It is intended only for exploring the data manipulation functions of CodeProbe and has no practical use. The following sections use examples based on Sample Program 2 to explore various commands to display data and data structures.

Source Code The following source code for Sample Program 2 is provided in the `sc:examples/debugger` drawer:

```
#include <stdio.h>

void main (void)
{
    struct X {
        int a;
        int b[3];
        int c;
    } x;

    struct Y {
        int a;
        int b;
    } y;

    typedef struct X *XPTR;

    struct W {
        int x;
        struct Y y[2];
        int z;
    } w;

    struct Z {
        char c;
        struct Y d;
        int e;
    } z;

    struct X3D {
        int a;
        int b[2][3][2];
        int c;
    } x3d;

    int i,j;
    int array[10];
    int *intptr;
```

```
long lng;
short shrt;

float fl;
double db;

char *string = "Hello, World";

/* Initialize data structures.  */
i = 33;
j = 44;
intptr = &i;

lng = 6456;
shrt = 89;

fl = 3.14;
db = 33.5;

array[0] = 5;
array[1] = 15;
array[2] = 25;
array[3] = 35;
array[4] = 45;
array[5] = 55;
array[6] = 65;
array[7] = 75;
array[8] = 89;
array[9] = 95;

x.a= 5;
x.b[0]=1;
x.b[1]=2;
x.b[2]=3;
x.c=11;

y.a=5;
y.b=6;

w.x=4;
w.y[0] = y;
w.y[1] = y;
w.z = 9;
```

```

z.c = 'r';
z.d = y;
z.e=3;

intptr = &j;

x3d.a=2;
x3d.b[0][0][0] = 41 ;
x3d.b[0][0][1] = 42 ;
x3d.b[0][1][0] = 43 ;
x3d.b[0][1][1] = 44 ;
x3d.b[0][2][0] = 45 ;
x3d.b[0][2][1] = 46 ;
x3d.b[1][0][0] = 47 ;
x3d.b[1][0][1] = 48 ;
x3d.b[1][1][0] = 49 ;
x3d.b[1][1][1] = 50;
x3d.b[1][2][0] = 51;
x3d.b[1][2][1] = 52;
x3d.c=90;

printf("%s!\n",string);
}

```

Running Program 2

To run sample program 2, compile, link, and load it into the debugger as follows:

1. Compile and link using the following command:

```
sc debug=sf link math=standard noautoregister data.c
```

2. Start CodeProbe by entering the following command:

```
cpr data
```

Examining Data and Data Structures

CodeProbe offers several commands that you can use to display the data referenced by your program. This section describes the following commands for examining data and data structures:

display

displays an object, area of memory, or the result of an expression

dump

examines the value of any variable in a program, an arbitrary range of memory, a register, or an expression

register

displays or modifies the current settings of the machine integer registers

watch

sets a variable or area of memory to be watched

wbreak

sets a breakpoint when a watched variable or area of memory changes

what is

determines the type and address of a variable or the fundamental C type to which a **typedef** identifier resolves.

The display Command

The **display** command displays C data structures and objects and can display an object or area of memory interpreted as a specified type of object.

The syntax of the **display** command is as follows:

```
d[isplay] (expression["format"] | string)[, ... ]
```

The *expression* parameter can be any C expression containing constants, variables, and function calls from the program being debugged except expressions involving the comma operator (**,**), the **?:** operator, the various assignment operators (**+=**, **-=**, and so on), or the prefix and postfix operators (**++** and **--**).

To override the default format, a format specification like those used with the **printf** statement can be used after each *expression*. The *format* parameter can contain up to two conversion operators for short and long types, but only one for all other types. No error checking is done on the format and strange results are possible.

If an address is specified as the *expression* parameter, and it is an address constant, then, in the absence of a *format* parameter, the *address* is assumed to be of type **char ***.

The *string* parameter can be any standard C string as described earlier in this chapter. The string is echoed in the Dialog window.

The **display** command can accept multiple arguments, each of the following form:

```
(expression["format"] | string)
```

The arguments must be separated by commas. Given this information, the following are examples of the **display** command:

```
> display i
> display 0xC80FF8
> display i "i is equal to %d"
> display a "a=%10.5e"
> display p->d[5] + myfunction(3)
> display x "X is %d", j "J is %d"
```

Before using Sample Program 1 in the examples shown in this section, enter the following command to execute all of the data initialization code:

```
> go 110
```

Now you can display the value of an integer variable as shown here:

```
> display i
33 (0x21)
```

To display its address, use the following command:

```
> d &i
0xC309B8
```

Note that register variables do not have an address; if you try to use the **and** (&) operator on a register variable, you will get an error message. This is why the example program was compiled with the **noautoregister** option.

It is equally simple to display an array element as follows:

```
> d *array
5 (0x5)
> d array[0]
5 (0x5)
```

You can display an entire array as follows:

```
> display array
{
  [0] = 5 (0x5)
  [1] = 15 (0xF)
  [2] = 25 (0x19)
  [3] = 35 (0x23)
  [4] = 45 (0x2D)
  [5] = 55 (0x37)
  [6] = 65 (0x41)
  [7] = 75 (0x4B)
  [8] = 85 (0x55)
  [9] = 95 (0x5F)
}
```

Structures are also easy to analyze using the **display** command as follows:

```
> d x
struct X {
  a = 5 (0x5)
  b = {
    [0] = 1 (0x1)
    [1] = 2 (0x2)
    [2] = 3 (0x3)
  }
  c = 11 (0xB)
}
```



```

> display z
struct Z {
  c = 'r' 114 (0x72)
  d = struct Y {
    a = 5 (0x5)
    b = 6 (0x6)
  }
  e = 3 (0x3)
}

```

Multidimensional arrays are displayed in a format that enhances analysis as follows:

```

> d x3d
struct X3D {
  a = 2 (0x2)
  b = {
    [0] = {
      [0] = {
        [0] = 41 (0x29)
        [1] = 42 (0x2A)
      }
      [1] = {
        [0] = 43 (0x2B)
        [1] = 44 (0x2C)
      }
      [2] = {
        [0] = 45 (0x2D)
        [1] = 46 (0x2E)
      }
    }
    [1] = {
      [0] = {
        [0] = 47 (0x2F)
        [1] = 48 (0x30)
      }
      [1] = {
        [0] = 49 (0x31)
        [1] = 50 (0x32)
      }
    }
  }
}

```

```

        [2] = {
            [0] = 51 (0x33)
            [1] = 52 (0x34)
        }
    }
    c = 90 (0x5a)
}

```

A conversion string can also be used as a *format* parameter:

```

> display i "i is equal to %d"
i is equal to 33

```

The dump Command

The **dump** command provides a simple way for you to examine the value of any variable in your program or to examine an arbitrary range of memory. The syntax of the **dump** command is as follows:

```

du[mp] [ variable | address | range ] [$] [style ...]

```

The *variable*, *address*, or *range* parameter specifies the location to be dumped.

The *style* parameter controls the format of the dump. For complete details on using the **dump** command, see the **dump** section in Chapter 9.

Dumping a Variable

In its simplest form, **dump** displays the memory at the address (the contents) of a variable. For example, the following command displays memory starting at the address of *i*:

```

> dump i
003C8466: 00000021

```

Note that the hexadecimal value **00000021** equals the decimal value **33**, the value of *i*. By default, **dump** displays memory in the hexadecimal format. The number to the left of the hexadecimal value, **003C8466**, is the address of *i*. The **dump** command always displays the addresses of the memory blocks displayed. To display memory in decimal format, use the **decimal** option.

```
> dump i decimal
003C8466:      33          0          0          0
003C8476:          0          0          0          0
003C8486:          0          0          0          0
003C8496:          0          0          0          0
```

The variable parameter can be any valid C variable. When a variable parameter is specified, the amount of memory dumped is determined by the size of the object (or in the case of a pointer, the size of the object to which it points.) However, in the case of a character pointer or an address constant, 64 bytes of data are always displayed as follows:

```
> dump string ascii
00394728:  H e l l o ,      W o r l d \0 \0 % s
00394738:  ! \n \0 \0 \0 \0 \0 \0 \0 \0 a \0 \0 \0 \0 \0
00394748:  \0 \0 \0 \0 \0 \0 \0 01 \0 \0 \0 \0 \0 \0 \0
00394758:  \0 \0 \0 01 \0 \0 \0 \0 \0 \0 01 \0 \0 \0 01
```

Dumping an Address

You can specify any valid address in the **dump** command, even addresses not under the control of your program. The following command specifies the address of the variable **i**. Note that the address parameter is preceded by a **0x**. Here is an example:

```
> dump 0x3c8466 decimal
003C8466:      33          0          0          0
003C8476:          0          0          0          0
003C8486:          0          0          0          0
003C8496:          0          0          0          0
```

Dumping a Range

When a **range** parameter is specified, exactly that amount of data is displayed, unless the size of the range is not perfectly divisible by the size of the specified type of data items. For example, the following command specifies to dump 7 bytes of integer data starting at the first address:

```
> dump 0x3c8466..0x3c846c decimal
0042D366:      33  -749469637
```

The dump actually includes an additional 1 byte of data to make up the second 4-byte integer. The **dump** command automatically rounds

the range up to the size necessary to correctly display at least all of the memory specified as the data type specified.

You can also specify ranges by indicating the number of elements of a data type to display, starting at a specified address or variable. The following command dumps the decimal value of the integer `i` and the next 12 bytes of memory (three additional integers of 4 bytes each):

```
> dump i L 4 decimal
003C8466:          33              2          41          4
```

The amount of memory that is displayed with a `range` option of this type depends on the size of the data type. A command with a range of four elements of type `decimal` will display 16 bytes. The default length of `float` for the `dump` command is 8, so a command with a range of four elements of type `float` will display 32 bytes.

Dumping a Pointer

Dumping pointers and memory pointed to by pointers can be confusing. In the Sample Program 2 (`data.c`) the following variables are defined and initialized:

```
int i;
int *intptr;

i = 33;
intptr = &i;
```

The following commands dump the contents of the variable `i`:

```
> dump i decimal
003C8466:          33

> dump intptr decimal
003C8466:          33              2          1          2
003C8476:           3              4          5          6
003C8486:           7              8          9         10
003C8496:          11             12         97 1927282688

> dump *intptr decimal
003C8466:          33
```

```
> dump &i decimal
003C8466:      33      2      1      2
003C8476:       3       4       5       6
003C8486:       7       8       9      10
003C8496:      11      12      97 1927282688
```

All of these commands are equivalent. The **dump** command treats variables that are addresses as already de-referenced. The **dump** command assumes that, if you specify a pointer, you want to look at what the pointer is pointing at. If you want to see the actual address contained in the pointer, you can look at the first address in the dump. You will note that all of the preceding dumps have **003C8466** as the address containing 33, which is the address of **i**.

To dump the actual address of the pointer variable, use the following command:

```
> dump &intptr
0064F676:    003C8466
```

The address of the pointer is the first number displayed. Sometimes the compiler automatically stores a variable in a register. If this has occurred for a variable, the preceding command will yield the following result:

```
> dump &intptr
^----& cannot be applied to registers or constants
```

Dumping Floating-Point Numbers

The **float** option is used with the **dump** command to specify the format of floating-point numbers. The default number of bytes dumped by the **float** option is 8. This is equivalent to type **double** under C. Note that there is no double format specifier. To specify a 4-byte **float**, equivalent to type **float** under C, use the **float4** option. Using the wrong format for a variable will cause the variable's value to appear incorrect. For example, compare the following commands and their results for the double variable **db = 33.5** and the float variable **fl = 3.14**:

```
> dump db float4
004103FE: +3.0117187
> dump db float
004103FE: +                33.5
```

```
> dump fl float4
00410406: +3.1400001
> dump fl float
00410406: +49.92001667991304
```

Dumping Integers

The **decimal** option is used with the **dump** command to specify the format of integers. The **decimal** option dumps 32-bit (4 byte) integers. This is equivalent to type **long** under C, and type **int** if the **shortinteger** compiler option was not used. Note that the **dump** command cannot distinguish between modules compiled with 16-bit integers (using the **shortintegers** option) and those compiled with 32-bit integers (using the default.) Using the wrong format for a variable will cause the variable's value to appear incorrect. The following commands will dump an **integer**, a **long**, and a **short**. Note the required format for the **short**:

```
> dump i decimal
00410436:          33
> dump lng decimal
00410402:        6456
> dump shrt decimal
00410400:       5832704
> dump shrt decimal2
00410400:         89
```

Dumping Characters and Strings

Characters and strings may be dumped in several ways. The simplest is to use the **ASCII** option as follows:

```
> dump z.c ascii
00410472:  r
> dump string ascii
003B1B88:  H e l l o ,      W o r l d \0 \0 % s
003B1B98:  ! \n \0 \0 \0 \0 \0 \0 \0 \0 @ \0 \0 F E4
003B1BA8:  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
003B1BB8:  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
```

When using the **ASCII** option, certain nonprintable characters appear as their usual C escape sequences:

```

\b  backspace
\f  formfeed
\n  new line
\r  carriage return
\t  horizontal tab
\v  vertical tab
\0  null byte (string terminator)

```

Other characters are represented by the two-digit hexadecimal codes.

You can also use the text option to display a block of data in character format. The text option displays the address of the data items on the left, the hexadecimal values in the middle, and the printable characters on the right.

```

> dump string text
003B1B88: 48656C6C 6F2C2057 6F726C64 00002573 Hello, World..%s
003B1B98: 210A0000 00000000 00004000 0046E420 !.....@...F.
003B1BA8: 00000000 00000001 00000000 00000000 .....
003B1BB8: 00000001 00000000 00000001 00000001 .....

```

The **dzero** command can also be used to display an ASCII text string terminated by a NULL character (**\0**) as shown here:

```

> dzero string
"Hello, World"

```

The string at that location is displayed in double quotes. Normally you would use either a character pointer or an array name as the argument to this command.

The register Command

The **register** command has two forms, with and without parameters as follows:

```

r[egister]

r[egister] [register[[]=] expression ]

```

The first form of the **register** command displays the contents of all the registers, while the second form results in the specified value being placed in the named register.

Displaying the Contents of Registers

If you use the **register** command without parameters, the current contents of the registers are displayed in hexadecimal followed by the current flag settings as shown here:

```
> register
d0=00000001 d1=00000002 d2=000003ED d3=00000000 d4=00000000
d5=00000000 d6=00000000 d7=00000000 a0=00C2C102 a1=000223B0
a2=00C25CC0 a3=00C2C106 a4=00C2BF08 a5=00000000 a6=00C0B9C8
sp=000229A8 pc=00C32E82 ccr=00000000 na pl nz nv nc
sort:\smain.c\main 110
```

Registers d0 through d7 and a0 through a6 are labeled and displayed. Register a7 is the stack pointer and is labeled **sp**. The program counter is labeled **pc**, and the condition code register is labeled **ccr** and is summarized by the two-letter flag abbreviations immediately following.

Note that the current source line number is displayed immediately after the display of the registers and flags.

Altering the Contents of Registers

The second form of the register command is similar to the **set** command. It allows you to alter the contents of registers. You can specify the value you want placed in a register as an expression. Here is an example:

```
> register
d0=00000001 d1=00000002 d2=000003ED d3=00000000 d4=00000000
d5=00000000 d6=00000000 d7=00000000 a0=00C2C102 a1=000223B0
a2=00C25CC0 a3=00C2C106 a4=00C2BF08 a5=00000000 a6=00C0B9C8
sp=000229A8 pc=00C32E82 ccr=00000000 na pl nz nv nc
sort:\smain.c\main 110
> register d7=5
> register a2=0x241a4a
> register
d0=00000001 d1=00000002 d2=000003ED d3=00000000 d4=00000000
d5=00000000 d6=00000000 d7=00000005 a0=00C2C102 a1=000223B0
a2=00241A4A a3=00C2C106 a4=00C2BF08 a5=00000000 a6=00C0B9C8
```



```
sp=000229A8 pc=00C32E82 ccr=00000000 na pl nz nv nc
sort:\smain.c\main 110
```

You also can set a register by using a program variable or other expression:

```
> display i
33 (0x21)
> register d3=i
> register
d0=00000001 d1=00000002 d2=000003ED d3=00000021 d4=00000000
d5=00000000 d6=00000000 d7=00000005 a0=00C2C102 a1=000223B0
a2=00241A4A a3=00C2C106 a4=00C2BF08 a5=00000000 a6=00C0B9C8
sp=000229A8 pc=00C32E82 ccr=00000000 na pl nz nv nc
sort:\smain.c\main 110
```

To set floating-point registers, refer to the **fregister** command in Chapter 9.

The **whatis** Command

The **whatis** command is used to determine the type of an object or the configuration of a type of object. It has two forms:

```
wha[tis] expression
wha[tis] (type)
```

The *expression* and *type* parameters can be any expression or data types, including structures and unions, provided by the C language.

Sample Program 2 defines the following objects:

```
struct X {
    int a;
    int b[3] ;
    int c;
} x;

typedef struct X *XPTR;
```

The **whatis** command yields the following:

```
> whatis x
auto at 00221FBC (20 bytes)
struct X x;
```

```

> whatis x.a
object at 00221FBC (4 bytes)
int a;
> whatis x.b[0]
object at 00221FC0 (4 bytes)
int
> whatis (XPTR)
typedef struct X {
    int a;
    int b[3];
    int c;
} *XPTR;
> whatis (int)
int
> whatis (struct X)
struct X {
    int a;
    int b[3];
    int c;
};

```

Notice that the **typedefs** are expanded in terms of the more fundamental types.

Watches and Watch Breaks

The **watch** commands allow you to set, list, disable, enable, and clear watches and watch breaks. A watch (also called a watch point) allows you to monitor a variable, an address, or a range of memory in order to see when the value of the variable changes or the area of memory is altered. A watch break is similar to a watch but acts as a breakpoint if the variable or area of memory changes.

Like breakpoints, watches and watch breaks remain in memory until you clear them or exit the debugger. You may restart the program being debugged and still retain these watch settings. This allows you to run through the program again and again without resetting the watches and watch breaks.

Watches and watch breaks are displayed in the Watch window. This window is updated when the debugger returns control to the user (as at a breakpoint), at which point the new values of the watched areas are displayed. You can open the Watch window with the F1 function key.

You also can display the values of the watched areas by using the **wlist** command. For example, if a watch break is set on the variable **tmp** in the function **swap** in Sample Program 1, then execution will

stop at line 6 of `swap` when the value of `tmp` changes. At this point, entering a `wlist` command will display the value of the watch as shown here:

```
sort:\swap.c\swap 6
> wlist
1! <static register> tmp : 0 (0x0)
```

The `1!` refers to the numbered entry in the watch list.

Watches provide a convenient way to watch memory on a continuing basis. It is often easier to examine the Watch window instead of entering a `display` or `dump` command after every step or breakpoint. You can also use the Memory window to keep track of an area of memory.

Watch breaks can help you locate problem areas with precision. Often a variable or block of memory may be altered or corrupted at an unknown point in a program's execution. It can be very difficult to pinpoint the exact location where the corruption took place. By setting a watch break on the variable and executing the program, the problem can be isolated quickly.

The watch and wbreak Commands

The `watch` command is used to set watches, while the `wbreak` command is used to set watch breaks. The syntax of these commands is as follows:

```
w[atch] variable | address | range [s[tatic] | d[ynamic]]
```

```
wb[reak] variable | address | range [s[tatic] | d[ynamic]]
```

The *variable*, *address*, or *range* parameter specifies the location of the watch or watch break. The optional `static` or `dynamic` flag specifies the way in which the debugger evaluates the location of the object being watched. (See Chapter 9 for more information.)

By default, watches are dynamic. This means that an expression being watched is re-evaluated whenever control is returned to the debugger. The `static` flag causes the address of a watched expression to be evaluated once when the watch is set.

Conversely, watch breaks are static by default. The `dynamic` flag is used to specify that the address of a watch break should be re-evaluated every time control is returned to the debugger.

With the `watch` command, the new value of the watched object is displayed in the Watch window whenever control is returned to you

(for example, at a breakpoint). If the Watch window is not open, you can use the `wlist` command as a way of determining the new value.

Watches (and watch breaks) can be set on structures or arrays by name. When this is done, the specification of the aggregate name is treated as an implicit range and a watch is set on that range. Assume the following declarations are in effect:

```
struct X
{
    int i;
    char a[20] ;
    double d;
} x;

char string[80] ;
```

A watch can be set on ranges as follows:

```
> watch x
> watch string[5] L 10
> watch &string[17].. &string[19]
```

Watches and watch breaks can be set on formal or automatic variables. In this case the watch is in effect only as long as the function-defining variable is executing. If you set a watch on a formal or automatic variable, the watch list displays the watch as not active when execution is taking place outside its defining function. When the function is re-entered, the watch is re-activated.

You should, however, use caution when using variables to specify a range. For example, suppose that `aut` is an automatic variable and `sta` is a static variable. Issuing the following command sets a watch on the memory between the address of `aut` and the address of `sta` as follows:

```
> watch &aut &sta
```

Of course, the address of `aut` could change a number of times as its function is entered, executed, and exited. In the case of such artificial ranges the debugger translates the symbolic names into their current absolute address equivalents and sets the watch on the absolute address range. It is this absolute address range that is displayed in the watch list. In general, this sort of translation from symbolic range to absolute range takes place when the two variables you use to specify

the range do not resolve to addresses within the same aggregate (structure, union, or array).

While watch breaks can be useful in monitoring program execution, they are expensive in terms of execution time. You should expect your program to execute hundreds of times more slowly when watch breaks are enabled. This is not true of watches since the information for these is updated only when control is returned to you by the debugger.

Watch breaks should be disabled whenever possible to speed execution. One way you can do this is to enable and disable a watch break dynamically depending on program conditions as shown here:

```
> wbreak i
> wdisable 1
> break sort 9 {we 1; go sort 20; wd 1; go}
```

This command sequence installs a watch break whose number in the example is assumed to be 1 and then it sets a breakpoint that automatically enables the watch break, executes up to a certain point, disables the watch break, and continues execution. In this way, you can force the watch break to be active only when a certain portion of your program is executing.

4 Setting Up the Debugging Environment

73	<i>Introduction</i>
74	<i>The debug= Options</i>
76	<i>Which Debugging Option To Use</i>
77	<i>Determining Source File Location</i>
78	<i>Choosing Command-line Options</i>
81	<i>CodeProbe Initialization</i>
82	<i>Setting and Showing Options within CodeProbe</i>
83	<i>Source Mode</i>
84	<i>Echo Mode</i>
84	<i>Instruction Bytes</i>
85	<i>Ignore Path</i>
85	<i>Case Sensitivity</i>
85	<i>Context</i>
86	<i>List Default</i>
86	<i>Unassemble Default</i>
86	<i>Radix Default</i>
86	<i>String Length</i>
86	<i>Array Dimension</i>
87	<i>Range Length</i>
87	<i>Maximum Bad Chars</i>
87	<i>Search Path:</i>
88	<i>Autoswap Mode</i>
88	<i>Catch New Devices</i>
88	<i>Step Into ResLib</i>
89	<i>Current Task</i>
89	<i>Catch New Tasks</i>
89	<i>Customizing Dialog Window Commands</i>
89	<i>The Alias Command</i>
92	<i>The Unalias Command</i>
92	<i>The Define Command</i>
94	<i>The Undefine Command</i>

Introduction

To use CodeProbe efficiently, you need to know how to set up the optimal debugging environment for your particular application. This

involves choosing the best debugging option to use, determining where to locate source files, and selecting necessary command-line options. This chapter discusses these and other topics pertinent to customizing your debugging environment.

This section also describes the command used to invoke CodeProbe, `cpr`. Selecting the correct options for CodeProbe can determine whether a debugging session will be productive and useful.

The debug= Options

The degree to which CodeProbe can access symbols defined in a module is determined by the compiler debugging option used in compiling that module.

The `debug=` option can be used with any of the following keywords to activate the debugging mode of the SAS/C Compiler:

line or **l**

generates the line number/offset table only.

symbol or **s**

in addition to the line number/offset table, generates full debugging information for those symbols referenced in the module.

symbolflush or **sf**

in addition to the line number/offset table, generates full debugging information for those symbols referenced in the module. Also forces the code generator to flush all registers to variable locations at line boundaries.

full or **f**

in addition to the line number/offset table, generates full debugging information for all symbols declared in the module even if there is no reference to them in the generated code.

fullflush or **ff**

in addition to the line number/offset table, generates full debugging information for all symbols declared in the program even if there is no reference to them in the generated code. Also causes the code generator to flush all registers at line boundaries.

Each of the `debug=` options is described in more detail later in this section.

You can mix modules compiled with different levels of debugging in the same program.

The `nodebug` option can be used to suppress the generation of debugging information. `nodebug` is the default.

The `addsym` option can be used to tell the linker to add symbol information to files that are compiled without debugging information. If a file was compiled with debugging information, `addsym` is not required.

The `debug=line` Option

Specifying the `debug=line` option causes the compiler to output the line number/offset table only. As a result of using the `line` option, the debugger can match the lines in a source file with the loaded code. This allows you to set breakpoints at source lines and step through the source code. In addition, some limited symbolic information is available for external variables and functions. Only the address of a symbol is available. There is no information about the symbol's attributes. For this reason, the debugger will attempt to display any variable as a `long` by default with the following warning message:

```
> display fahr
accessing extern with unknown attributes - assuming long
1101004800 (0x41A00000)
```

To display a variable in its proper form, use a type cast with the `display` command as follows or use one of the `dump` commands.

```
> display (float) fahr
20
```

See Chapter 9, “Commands and Built-in Functions,” for details on the `display` and `dump` commands.

While the `line` option provides limited information to the debugger, it is useful with large programs or when disk space is limited because it produces much smaller object files. The `line` option also provides all the information needed by the disassembler, `omd`.

The `debug=symbol` Option

Specifying `debug=symbol` causes the compiler to output full debugging information for all global and local variables, all functions, and all types referenced in the module, including `structure`, `union`, `enum`, and `typedef` types. If a module is compiled with this option, you can set breakpoints at source lines, display or alter data symbolically, set watches on variables or areas of memory, and list source code. The `display` command will properly format all data declared in the module.

Debug information for a structure or union definition is generated only if it is used in a `typedef` declaration or as either a global or local variable. That is, if the structure or union is defined in a header file that is included by your program, but the tag associated with that structure or union is not used by your program, then no debug information will be generated for that tag. Most source modules use

numerous include files that contain definitions that are never actually used. By limiting debug information to include only referenced definitions, the output module size is greatly reduced.

The compiler will often manipulate a variable's contents in registers over several lines of code without replacing the results in the variable. If a breakpoint is reached in this section of code, the debugger will display the value in memory, not in the register. If you suspect that the debugger is not displaying the proper values, the code can be inspected in mixed mode, or the file can be recompiled with the **symbolflush** option. Note that register variables (automatic and formal variables that are only assigned to registers) will always be found in their proper register.

The **debug=symbolflush** Option

Specifying **debug=symbolflush** causes the compiler to output full debugging information for all global and local variables, all functions, and all types referenced in the module, including **structure**, **union**, **enum**, and **typedef** types. It does not generate debugging information for types that are defined but not referenced in the module.

The **symbolflush** option also causes the code generator to flush all registers to variable locations at C source line boundaries. This can be useful because the debugger takes the value of a variable from its memory location, not a temporary register. While this option is recommended for use with the debugger, it causes inefficient code generation. Any module compiled with this option should be recompiled before it is placed into production.

The **debug=full** Option

The **debug=full** option is similar to the **debug=symbol** option except that information for all declarations and definitions is generated including those that are defined but not referenced in a module.

The **debug=fullflush** Option

The **debug=fullflush** option is similar to the **symbolflush** option except that information for all declarations and definitions is generated including those that are defined but not referenced in a module.

Which Debugging Option To Use

In order to make full use of the debugger's features, you should always use either the **symbol**, **symbolflush**, **full**, or **fullflush** option. If the compiler keeps the value of a variable in a register, as it may in order to eliminate unnecessary loading and storing of values in its

registers, then the true value of a variable at any given time may be in a register rather than in the variable's memory location. As a consequence, the value of the variable displayed by the debugger may not be its correct value. The **symbolflush** and **fullflush** options force the correct values of variables to be in the memory locations of the variables at line boundaries but at the cost of generating extra code required to store, and perhaps reload, the values.

Only the **symbolflush** and **fullflush** options affect the quality of code generated by the compiler. The other options do not affect code generation and differ only in the amount of symbolic information they pass on to the linker. If you want to debug exactly the program you intend to use, you should use **symbol** or **full** rather than **symbolflush** or **fullflush**, and you should debug in mixed mode so that you can see the machine instructions corresponding to your C source lines. The **opt source mixed** command is used to specify mixed mode. When you do this, you will be able to tell if the value of a variable is being kept in a register rather than being flushed to memory, and you can then use the **register** command rather than the **display** or **dump** commands to determine the variable's value. See Chapter 9 for more details about these commands. To produce the final version of your program, use the **stripdebug** option of **slink**. For example, the following command will strip debugging information from an existing executable file, *source-filename*, and place it in a second file, *destination-filename*:

```
slink from source-filename to destination-filename stripdebug
```

If you are not concerned with debugging production quality code, you should use the **symbolflush** option, since the difference in generated code will have little effect. Even if you are attempting to create a piece of production software, you should begin your debugging by using the **symbolflush** option and then recompile with the **symbol** option when most of your bugs have been detected and repaired.

Determining Source File Location

If you have compiled your program with any of the **debug=** options, CodeProbe reads the name of the source file from the program being debugged. By default, this is the same name you passed to the compiler with the **SC** command. If you specify a full or partial pathname for your source file, the pathname is included. You can change the filename specification in the program's debugging information by using

the **sourceis** option when you compile your program. Refer to Chapter 8, “Compiling and Linking Your Program,” in *SAS/C Development System User’s Guide, Volume I* for more details. You may want to direct CodeProbe to look for source files on a different disk or at a different location on the same disk by using the **opt search** command described later in this section.

If CodeProbe cannot find the source file for the function it is currently executing, it displays the following message:

```
Error: error opening source file filename
```

Your source code will be displayed as long as you have used some debugging option, other than the **nodebug** option, with the compiler.

The CodeProbe **source** command can be used to override the filename specified in the debugging information. This new association is remembered by CodeProbe throughout the current session. This command can prove useful when the name of the source file has been changed since the program was compiled.

Choosing Command-line Options

The CodeProbe environment can be customized by including command-line options when invoking the debugger. The format of the command is as follows:

```
cpr [cpr-options] application-name [application-arguments]
```

For information on how to use command-line options with the program you are debugging, see Chapter 1, “Introduction to CodeProbe.” If you invoke CodeProbe from the Workbench screen, you can use command-line options by adding them as tool types to the **debug** icon created by **scsetup**. To add tool types, click on the icon, then select Info from the Workbench menu under AmigaDOS Version 1.3, or Information from the Icons menu under AmigaDOS Version 2.0.

You can use the following command-line options when invoking CodeProbe:

-buffer

specifies the size of the Dialog window buffer. By default the Dialog window saves the last 4096 bytes displayed so that you can scroll

backwards and review output. This size can be increased or decreased with this option. The syntax of the option is as follows:

```
cpr -b[uffer] size
```

The *size* parameter specifies the size of the buffer in bytes.

-cli

instructs CodeProbe to invoke the application as a CLI (Shell) process. CodeProbe passes arguments to the application through the normal command line interface. This option is the default if CodeProbe is invoked from a Shell. The option is necessary if CodeProbe is invoked from the Workbench screen, and you want your program to run as if invoked from a Shell.

-command

executes debugger commands at startup. The commands are executed after `go main` if the `-startup` option is not specified or after the profile script if it is specified. Profile scripts are discussed later in this section.

For example, the commands

```
cpr -command "proceed; display fahr" program-name
```

would execute to main, step over 1 line of code, and display the variable `fahr` in the Dialog window before giving control to you.

-i

sets up a screen in interlace mode. By default, CodeProbe opens a new screen using the specifications set up by Preferences for Workbench screens. To force a screen to be opened in interlace mode, include the `-i` option before typing the application command name.

-line

starts CodeProbe in line mode. Line mode causes the debugger to run in the current Shell window. The equivalent of the Dialog window is provided, with no other windows available. Line mode is probably only useful for running from a script file, redirecting output using the Shell redirection facility, or running from a remote terminal over a communications port.

-noprofile

suppresses execution of the CodeProbe startup file. When CodeProbe is invoked, it automatically executes a script file named `cprinit`, which must be located in either the current directory or

ENV:sc. If the startup file is found in the current directory, **ENV:sc** is not searched.

-screen

instructs CodeProbe to open its windows on the named public screen. The syntax of the command is as follows:

-sc[reen] *name*

By default, CodeProbe creates a new public screen when it is invoked. If this option is specified, CodeProbe looks for an existing public screen with the given name, and if one is found it uses that screen to display its windows. If a screen with the given name does not exist, a new one is created with that name. To use the Workbench screen, specify **-screen workbench**. Public screens are a feature of Intuition 2.0. For users of earlier versions of Intuition, this option is not generally useful. However, specifying **-screen workbench** causes CodeProbe to use the Workbench screen on older releases.

-startup

suppresses the automatic **go main** that is normally executed by the debugger on startup. This option is useful if your application redefines **_main** or **c.a** so that no main function exists, or if you want to step through such code. If this option is used, no initialization of any kind will have been performed by the application process before control is given to you.

If the **quit**, **start**, or **restart** commands are invoked and an application process has not exited, the debugger normally calls **exit** to clean up any process resources that may not have been freed. However, if the **-startup** option was used, **exit** will not be called.

-w

runs CodeProbe on the Workbench screen. This is the same as specifying **-screen workbench**.

-wb

instructs CodeProbe to invoke the application as a Workbench process. It passes arguments to the application through a Workbench startup message. This option is the default if CodeProbe is invoked from a Workbench icon. The option is necessary if CodeProbe is invoked from a Shell, but you want the application to run as if invoked from the Workbench screen.

-wdialog, -wregister, -wsource, -wwatch

specifies start-up window coordinates for the Dialog, Register, Source, and Watch windows, respectively. Window coordinates are measured in character positions. A width or height of 0 causes the window to extend to the screen border on the right or bottom, respectively. The option names can be abbreviated to two letters:

-wd[ialog]	<i>left</i>	<i>top</i>	<i>width</i>	<i>height</i>
-wr[egister]	<i>left</i>	<i>top</i>	<i>width</i>	<i>height</i>
-ws[ource]	<i>left</i>	<i>top</i>	<i>width</i>	<i>height</i>
-ww[atch]	<i>left</i>	<i>top</i>	<i>width</i>	<i>height</i>

Here is an example:

```
cpr -ws 0 1 50 12 -wd 0 13 50 0 ftoc
```

Note that the Register and Watch windows will not be displayed on startup; however, when they are opened by pressing the appropriate function key (either F1 or F4), they will open to the coordinates specified on the command line.

-.

suppresses the copyright banner that is displayed at startup.

CodeProbe Initialization

Since you will probably want CodeProbe to start the debugger with the same options most of the time, CodeProbe allows you to create a debugger command script file that is executed each time the debugger is invoked. The profile script, named **cprinit**, is executed after the initial **go main** but before any commands specified with the **-command** option are executed. CodeProbe first looks for **cprinit** in the current directory and then in the **ENV:SC** directory. Only one **cprinit** script will be run; therefore, if a script is located in both directories, the one in the current directory is used.

The following are examples of scripts displayed from the Shell window using the AmigaDOS **type** command:

```
1> type cprinit
/* This is the debugger startup profile */
```

```

opt search dh0:mysource, df0:mysource, ram:
opt radix h
opt ibytes off
1> type env:sc/cprinit
/* This is a debugger profile script for a project */
opt source mixed
proceed
display argc, argv

```

Note that C style comments can be placed in profile scripts as long as they exist on a single line.

Setting and Showing Options within CodeProbe

In addition to the command-line options discussed above, a number of options can be set from within CodeProbe to customize the way information is displayed or handled. Most of these options are modified with the `opt` command or through the Options menu. To see the default options while in the debugger, activate the Options menu or execute the `opt` command without any parameters as follows:

```

> opt
Source mode.....c
Echo mode.....Off
Instruction bytes...Off
Ignore path.....Off
Case sensitivity...Off
Context.....2 lines
List default.....6 lines
Unassemble default..4 lines
Radix default.....Decimal
String Length.....128
Array Dimension.....20
Range Length.....64
Maximum Bad Chars...3
Search path:
Autoswap mode.....Off
Catch New Devices...No
Step into ResLib....No
Current Task.....sort at 0001DC20
Catch New Tasks.....No

```

A CPR initialization script can be used to change options automatically when you invoke the debugger. See “CodeProbe Initialization” earlier in this chapter for more information.

Source Mode There are three source modes: C, assembly, and mixed. The current source mode can be determined by means of the `opt` command or by activating the Options menu. It can be changed by means of the `opt source` command or the Source selection in the Options menu. C source mode is the default.

The syntax of the `opt source` command is as follows:

```
op[t] so[urce] c | a[sm] | m[ixed] | n[ext]
```

Repeatedly choosing `opt source next` cycles through the three possible source modes.

C Mode

In C mode, CodeProbe displays C source lines when the `trace` or `proceed` commands are executed or when a breakpoint is triggered. To change the current source mode to C mode, enter the following command:

```
> opt source c
```

While in C source mode, you cannot single step by assembly instruction.

Assembly Mode

In assembly mode, disassembled code is displayed in the Source window. Single `trace` or `proceed` commands will step by assembly instruction.

To change the current source mode to assembly mode, use the following command:

```
> opt source asm
```

Mixed Mode

In mixed mode, both assembly instructions and C source lines, if available, are displayed in the Source window. For each C source line displayed, the corresponding assembly instructions are displayed as well.

To change the current source mode to mixed mode, enter the following command:

```
> opt source mixed
```

In mixed mode, you can take advantage of the two different forms of the **trace** and **proceed** commands in order to step by either C source line or assembly instruction.

Echo Mode When echo mode is **on**, commands are echoed in the Dialog window before being executed. This option is useful if you want to execute debugger command files and see the command lines as they are executed along with their output. Commands that are invoked by menu selections or double-clicking the mouse are also displayed as they are executed. By default, this option is set to **off**. It is not needed while entering commands in the Dialog window because command lines are displayed as they are entered anyway. The syntax is as follows:

```
op[t] e[cho] [ on | off ]
```

Instruction Bytes The **ibytes** option affects the display format of disassembled code in the Source window and the output of the **unassemble** command. If the option is set to **on**, the second field of the disassembly contains a hex dump of the instruction. The following fields contain the op-code and its operands. If the option is set to **off**, the hexadecimal dump is suppressed. The syntax is as follows:

```
op[t] ib[ytes] [ on | off ]
```

Here is an example:

```
> opt ibytes on
> unassemble 7
main:
    7 (
0025F950 48E70130          MOVEM.L    D7/A2-A3, -(A7)
0025F954 BFEC0004          CMPA.L     0004(A4), A7
0025F958 65001BD6          BCS        00261530
> opt ib off
> unassemble 7
```

```

main:
    7 {
0025F950      MOVEM.L    D7/A2-A3,-(A7)
0025F954      CMPA.L     0004(A4),A7
0025F958      BCS        00261530

```

Ignore Path If the `ignorepath` option is set to `on`, CodeProbe ignores the directory part of the source filename. In this case, CodeProbe looks in the current directory for the source file. By default, `ignorepath` is set to `off` and CodeProbe uses the entire pathname when searching for the source file. The syntax is as follows:

```
op[t] ig[norepath][ on | off ]
```

This option is useful in conjunction with the `opt search` command to override the source filename specified in the object file. For more information, see the `SOURCEIS` option in Chapter 8, “Compiling and Linking Your Program,” of the *SAS/C Development System User’s Guide, Volume I: Introduction, Compiler, Editor*.

Case Sensitivity The `case` option controls case sensitivity for the `search` command. The `search` command is used to search for a string in the Source window. When `case` is set to `on`, a case-sensitive search is made whenever a `search` command is executed. The syntax for the `case` option is as follows:

```
op[t] cas[e][ on | off ]
```

Context As you proceed through an application program by tracing and breakpointing, CodeProbe always keeps the current source line or assembler instruction displayed in the Source window. CodeProbe attempts to keep at least a minimum number of context lines of code displayed above and below the current line. The number of these context lines can be controlled by the `context` option. Note that, in mixed mode, it is not always possible to keep the right number of lines above or below, but the source code line will always be displayed and the assembler instruction will be displayed if possible. The default number of context lines is 2. The maximum number of context lines possible is limited by the number of lines in the Source window. The syntax is as follows:

```
op[t] co[ntext] integer
```

List Default The `list` option determines the number of source lines displayed by the `list` command. The `list` command is discussed in Chapter 9. The default value for this option is 6. The syntax is as follows:

```
op[t] l[ist] integer
```

Unassemble Default The `unassemble` option determines the number of instructions displayed by the `unassemble` command. The default value for this option is 4. The syntax is as follows:

```
op[t] u[nassemble] integer
```

Note that the default value is only used when no source file is available. When a source file is available for a section of code, `unassemble` displays the disassembly for a single source line by default.

Radix Default The `radix` option controls the default input type for constants. The choices are decimal and hexadecimal. The default is decimal. When set to `hex`, this option can be used to avoid having to type the initial `0x` on hexadecimal constants. The syntax is as follows:

```
op[t] rad[ix] [d[ecimal]| h[ex]]
```

String Length The `strlen` option controls the maximum number of bytes that are displayed when a character string is displayed with the `display` or `dzero` commands. If the string contains bad characters, fewer may be displayed. The default is 128. The syntax of the `strlen` option is as follows:

```
op[t] st[r]len integer
```

The *integer* parameter specifies the number of bytes.

Array Dimension The `arrdim` option specifies the maximum number of array elements that will be displayed by the `display` command. The syntax for the `arrdim` option is as follows:

```
op[t] ar[r]dim integer
```

The *integer* parameter specifies the maximum number of array elements. The default value is 20. Arrays with *integer* or fewer

elements are displayed in their entirety. Larger arrays will have their first *integer* elements displayed followed by an ellipsis (. . .).

Range Length The **rangelen** option controls the default size of memory dumps when no other source of information is available, for example, when dumping an absolute address. The **rangelen** option is used when commands such as **display**, **dump**, or **watch** have only an *address* parameter. The **rangelen** option specifies the number of elements after the address to display, dump, or watch. The syntax for the **rangelen** option is as follows:

```
op[t] ran[gelen] integer
```

The *integer* parameter specifies default range size in number of elements of a type to be displayed. The default value is 64.

Maximum Bad Chars The **badchar** option determines how many nonprintable characters must be encountered before the **dzero** or **display** command stops printing a string. The syntax of the **badchar** command is as follows:

```
op[t] b[adchar] integer
```

The *integer* parameter specifies the number of nonprintable characters allowed. The default value is 3. A 0 indicates no limit.

Search Path: The search path is used to find the source modules used when compiling the application. When a module is compiled with one of the debug options, the full pathname of the source module is saved in the debug information. CodeProbe always looks for the source module in its original location first, then it looks in the current directory, and finally, it walks through each of the directories specified in its search path in order. If you have moved the source module to some other location, use the **opt search** command to add the new location to your search path.

The **opt search** command takes four forms:

```
op[t] se[arch]
```

```
op[t] se[arch] dir[, dir[...]]
```

```
op[t] se[arch] + dir[, dir[...]]
```

```
op[t] se[arch] - dir[, dir[...]]
```

The first form lists the current paths in the search path list, which is empty by default. The second form sets the search path to the list of directories on the command line. The third form appends the list of directories on the command line to the current search path. The last form deletes the list of directories from the current search path. This list is empty by default. Here are some examples:

```
> opt search c:, df0:, dh0:mysource
> opt search + df1:test
> opt search - df0:, c:
```

The **search** option cannot be set from the Options menu.

Autoswap Mode

When in autoswap mode, the screen containing the application's output is pushed to the front each time CodeProbe gives control to the application. When a breakpoint is reached, the debugger screen is again pushed to the front. If you are single stepping through source code, the switching of screens will probably appear as a brief flash. If the autoswap mode is set to **off**, which is the default, the application screen will not be pushed to the front. The autoswap feature can be set to **on** or **off** with the **opt** command. The syntax is as follows:

```
op[t] au[toswap] [ on | off ]
```

Autoswap mode is particularly useful when debugging programs that require input from the keyboard. The application's screen is displayed whenever input is required.

Catch New Devices

When a device is opened for the first time a new task may be created. If the process that opened the device is under debugger control, then CodeProbe will catch the new task provided the Catch New Devices option is set to **on**. The default setting for this option is **off**. The syntax for the **devices** option is as follows:

```
op[t] d[evices] [ on | off ]
```

Step Into ResLib

The Step Into ResLib option allows the user to decide whether to allow CodeProbe to step into resident libraries while tracing or running an application with watch breaks. This option should remain set to **off** (default) unless you want to debug your own resident libraries. Tracing into system libraries such as **Exec** or **Intuition** can lead to problems. If this option remains off, it is safe to run with watch breaks on while stepping over system resident libraries. However, you should

disable all watch breaks before stepping into a ROM-resident library routine. The syntax for the **reslib** option is as follows:

```
op[t] res[lib] on | off
```

Current Task The current task is displayed by entering the **opt** command with no parameters. This information cannot be displayed through the Options menu. The following line indicates that a program named **sort** is the current task and that execution has stopped at address 0001DC20.

```
Current Task.....sort at 0001DC20
```

The current task cannot be changed with the **opt** command; however, you can use the **jump** command to change the execution point.

Catch New Tasks Whenever a new task is started by a task under CodeProbe's control, the new task is also under CodeProbe's control by default. This does not apply to tasks started by a call to the **OpenDevice** function. The **catch** option enables you to turn task catching **on** or **off**. The syntax of the **catch** command is as follows:

```
op[t] ca[tch] [ on | off ]
```

Customizing Dialog Window Commands

In order to provide CodeProbe with a more versatile command interface, the **alias** and **define** commands can be used to customize the Dialog window commands. Aliases enable you to redefine or customize commands. This capability is similar to the aliasing capabilities found in the Shell. Defines provide CodeProbe with a macro facility modeled after the C language's preprocessor **#define** facility.

The Alias Command The **alias** command is used to create new aliases, display the definition of an existing alias, or display a list of all currently existing aliases. The syntax for the **alias** command is as follows:

```
a[lias] [name[definition]]
```

When entered with no parameters, the **alias** command displays the list of all currently defined aliases, with the alias listed on the left

and the associated command on the right. The following example shows the list of aliases already defined for you by CodeProbe:

```
> alias
da                dump $* $ ascii1
db                dump $* $ hex1 text
dc                dump $* $ dec1
dd                dump $* $ float8
df                dump $* $ float4
dffp             dump $* $ ffp4
di                dump $* $ dec4
dl                dump $* $ dec4
dp                dump $* $ hex4
ds                dump $* $ dec2
dw                dump $* $ hex2
```

The definition for an existing alias is displayed if the **alias** command is entered with only the **name** argument. The following **alias** command causes the definition for the **dffp** alias to be displayed:

```
> alias dffp
dffp             dump $* $ ffp4
```

Alias names are only expanded when they occur at the beginning of a command line. An alias definition can be a simple textual substitution or it can contain positional parameters to be expanded at execution time.

Positional parameters are specified as **\$1**, **\$2**, **\$3**, and so on. **\$0** represents the alias itself. **\$*** represents all command-line parameters. If no positional parameter is specified in a definition, all parameters are placed after the expanded definition until it encounters a semicolon (;) or an end of line. To define an alias that expands into more than one command, enclose the definition in either double quotes (") or curly braces ({}). If the command string must contain double quotes, curly braces must be used or the quotes must be escaped. A backslash (\) can be used to continue the line.

The following are examples of the `alias` command:

```
> alias print display
> alias dw
dw                                dump $* $ hex2
> alias look2 "g $1; where; g $2; where"
> alias printgo {print "vars:  ", $*; go}
```

After entering these `alias` commands you could enter the following command lines:

```
> look2 sort swap
sort:\sort.c\sort 20
  1* In routine sort:\sort.c\sort 20
  2 Called from sort:\smain.c\main 13 (+0xC)
  3 Called from 0xC3162E
sort:\swap.c\swap 5
  1* in routine sort:\swap.c\swap 5
  2 Called from sort:\sort.c\sort 26 (+0x12)
  3 Called from sort:\smain.c\main 13 (+0xC)
  4 Called from 0xC3162E
> print tmp
10 (0xA)
> dw x
0036c4a4: 0000 0001
> go sort
> printgo i, p[i]
vars:  0 (0x0)  0 (0x0)
Program exited with code 0.
```

The `alias` definitions in the previous example expand to the following commands:

```
display tmp
dump *y hex 2
g sort; where; g swap; where
display "vars:  ", i, p[i]; go
```

If at least one parameter is specified in a definition, all unused parameters are thrown away in the expansion. Any unspecified parameter results in an empty expansion. For example, assume the

alias given previously for `look2`. You can enter the following command lines:

```
> look2 func1 func2 func3
> look2 func1
```

These command lines are expanded as follows:

```
g func1; where; g func2; where
g func1; where; g ; where
```

Aliases can be nested in the definitions of other aliases. If the debugger detects a circular reference between aliases, it will expand until it detects the cycle. If an alias references itself, the reference will not be expanded. For example, `alias where where args` does not cause an infinite loop.

Aliases can be used to redefine any native CodeProbe command except `alias` and `unalias`. To reference a command that has been aliased, escape the command name with a back-tick character (```). For example, the following sequence defines aliases for the `set` and `enter` commands so they function as they did in CodeProbe Version 5.10:

```
> alias set opt
> alias enter `et
```

Note the back-tick character used to escape the `set` command.

The Unalias Command

The `unalias` command removes entries from CodeProbe's list of aliases. The syntax for the `unalias` command is as follows:

```
unal[ias] name
unal[ias]*
```

The *name* parameter should match the alias name to be removed. The asterisk (*) is used to remove all aliases.

The Define Command

The `define` command provides a more general macro text substitution facility than the `alias` command. Just as with the C preprocessor's definitions, macros defined in CodeProbe are expanded anywhere within a line of text unless hidden inside of double quotes. CodeProbe will not expand macro definitions that have been escaped

by a back-tick character. The syntax for the **define** command is as follows:

```
[#]def[ine] name [definition]
[#]def[ine] name (parm[,parm...]) [definition]
```

The *name* parameter specifies the text to be replaced, and the *definition* parameter is the substitution text. In the second form, parameters are specified exactly as in the C language. Using the **define** command with no arguments displays all macro definitions currently recognized.

Some examples are as follows:

```
> define PI 3.14159
> display PI
3.14159
> define ISEQUAL(a,b) (a == b)
> display ISEQUAL(1,5)
0 (0x0)
> expand ISEQUAL(1,5)
(1 == 5)
```

To define a symbol to nothing, omit the **definition** parameter as shown in the following command:

```
> define FOO
```

One difference between the C preprocessor **#define** capability and CodeProbe's is that the debugger treats **define** as a command statement, meaning that it will stop parsing the **define** when a semicolon is reached. To have CodeProbe treat a semicolon as part of a definition, precede it with an escape character as follows:

```
\;
```

The **define** command is intended to be used to define expressions. Although you can use the **define** command to create new commands, you should use the **alias** command to create custom commands.

The Undefine Command

The `undefine` command removes a definition from CodeProbe's definition list. Compiler generated symbols may not be undefined. The syntax for the `undefine` command is as follows:

```
und[efine] name  
und[efine] *
```

The first form removes the macro with the specified name from the definition list. The second form removes all definitions from the list.

5 Working in Cross-Development Mode

95	<i>Introduction</i>
95	<i>Why Debug in Cross-Development Mode?</i>
96	<i>Using the Cross Debugger</i>
96	<i>Preparing to Use the Cross Debugger</i>
96	<i>Running the Cross Debugger</i>

Introduction

Version 6.0 of the SAS/C Development System comes with a new debugging capability: cross-debugging. This means that you can have your program run on one Amiga system, and debug it from another Amiga system. This is extremely useful when debugging a program that shuts down WorkBench, partially takes over the machine, or makes heavy use of graphics.

When running in cross-development mode, CodeProbe communicates with a small kernel that controls the application via a serial communications port or over a network.

Why Debug in Cross-Development Mode?

There are several reasons why you may want to debug in cross-development mode:

- It is easier sometimes to see and send input to both the application and debugger if they have their own separate monitors and keyboards.
- Some applications use low-level graphics control, which takes control away from Intuition. Since CodeProbe's windowing system is built on top of Intuition, you cannot communicate with it while such an application is running.
- With very large applications in tight memory situations, there may not be enough memory available for both CodeProbe and the application. The CodeProbe cross-debugger kernel that runs on the machine with the application is small (less than 30K). It allows the application to use most of the available memory.
- Disk space limitations may make it difficult to have the debugger, all the necessary source files, and a copy of the application with debug

information available on the same machine along with all the files needed by the running application.

- If the program being debugged crashes, you can continue to work on the machine with the source code while it is rebooting.

Using the Cross Debugger

In the following discussion, the machine containing the source code and the cross debugger (CPRX) is referred to as the *host machine*. Your actual debugging session occurs on the host machine. The machine on which the application will run under the kernel's control is referred to as the *target machine*.

The cross debugger and kernel communicate with each other over a serial link or via a named pipe. Using named pipes, you can debug your program from an Amiga system connected to the target machine via a network.

Preparing to Use the Cross Debugger

To use the cross debugger, you will need two Amiga machines. You can use a NULL modem cable to connect the machines, or you can even debug over the phone, if both Amiga machines are connected to modems. If you use the serial port, use the serial icon in the Preferences menu to select the highest baud rate acceptable. With a NULL modem cable, 19200 works fine. Over the phone you probably will need to specify the maximum speed accepted by your modems.

Running the Cross Debugger

Two files must be installed on the target machine: the kernel (CPRK), and the application program that is to be debugged, unless the `-x` option is used. CPRX can automatically strip the application program of debug information and install it on the target machine if the `-x` option is used.

CPRK provides direct control of the application program on the target machine. It communicates with CPRX, which is running on the host machine, over a serial line or through a pipe across a network. CPRK displays its copyright information and diagnostic output in a CLI window. CPRX runs like CodeProbe, except that the program being debugged runs on another machine.

On the host machine, you need the source to your project and the executable file for the program you want to debug. The executable file should have been built with debug information.

Debugging a program in cross-development mode is a four-step process:

1. Start the kernel (CPRK) on the target machine.
2. Start the cross debugger (CPRX) on the host machine.

3. Debug your application program.
4. Enter the `finish` command in CPRX to terminate CPRK and CPRX.

Note: When debugging over a serial port, you must start CPRK before starting CPRX. When using named pipes, the order does not matter.

Defining the Communications Parameters

CPRX and CPRK take the following command-line options that define and set parameters for the communications medium:

-pipe [*name*]

specifies that a named pipe is to be used instead of the serial port. If *name* is not specified, `pipe:cpr` is used. CPRX or CPRK will open two pipes using *name* as a base, `name_d` and `name_k`.

To communicate successfully over a network, either CPRX or CPRK must be invoked with a pipe filename referring to a pipe on the other machine. If your network allows you to refer to a pipe device on an attached machine by specifying

```
net:pipe/<pipename>
```

you could invoke CPRK on the target machine with the option

```
-pipe net:pipe/cpr
```

and invoke CPRX on the host machine with the `-pipe` option and no parameters. This method works with the popular freely redistributable network parnet, written by the Software Distillery.

-device [*name*]

specifies that the named device should be opened for communications if `-pipe` was not specified. The default is `serial.device`.

-unit [*number*]

specifies the unit number of the communications device if `-pipe` was not specified. The default is zero (0).

-speed [*number*]

specifies the baud rate of the communications device if `-pipe` was not specified. The default is the value set in preferences.

Starting the Kernel (CPRK)

Start the kernel on the target machine with the following Shell command:

```
cprk [options]
```

The only options CPRK accepts are the communications configuration options previously listed. CPRK cannot be invoked from the Workbench screen.

Once invoked, CPRK acts as a server, waiting for a request to start a debugging session from a CPRX invocation on another machine. Note that if you are using the serial device, CPRK must be invoked before CPRX is invoked on the host machine.

Starting the Cross Debugger (CPRX)

Start the cross debugger on the host machine with the following Shell command:

```
cprx [options]target-executable-filename [program-options]
```

CPRX cannot be invoked from the Workbench screen.

The *target-executable-filename* parameter is the name of the program to be debugged. By default, CPRX and CPRK look for the program in the current directory of both the host and the target machines. If the specified executable name is an absolute path, CPRX and CPRK look in that location on both machines. If the executable file is located in different places on the target and host, then use the **-symfile** option on CPRX to specify where the executable file resides on the host machine. CPRK will still look in the location specified by *target-executable-filename*. Anything on the command line after the *target-executable-filename* is passed to the target program as command-line arguments.

CPRX can be invoked with any of the options available to the native debugger (CPR) or with any of the communications configuration options previously listed. In addition, CPRX accepts the following options:

-symfile *filename*

specifies the location of the executable file on the host machine. Use this option if the location is not the same on the host and target machines.

-nok[check]

turns off checksum checking. By default, CPRX and CPRK examine a checksum on the executable files on the host and target machines

to verify that both sides are using the same executable file. If the checksum fails, CPRK does not load the program.

-x

strips all debugging information from the host's version of the executable file, copies the stripped version to the target machine, and tells CPRK to invoke the just-transmitted version. This allows CPRK to run constantly without changing disks or rebooting (as long as your program does not crash).

Example 1

Suppose you want to start a cross-debugging session to debug a program named **myprog** using the serial port at 19200 baud. Assume that the program to be debugged resides in the current directory on the host machine, so use **-x** to copy it over. The following commands can be entered on the target and host machines:

target machine

```
cprk -speed 19200
```

host machine

```
cprx -x -speed 19200 myprog
```

Example 2

Suppose you want to start a cross-debugging session to debug a program named **myprog** using **parnet** and named pipes. Assume that the program to be debugged resides in **TEMP:** on the target machine and in **work:test** on the host machine. The following commands can be entered on the target and host machines:

target machine

```
cprk -pipe net:pipe/cpr
```

host machine

```
cprx -pipe -symfile work:test/myprog temp:myprog
```

Debugging Your Application

Once started, the cross debugger behaves just like the normal debugger, except entering **CTRL-C** in the CPRX Dialog window does not always stop the program executing on the target machine. All interaction with the debugger occurs on the host machine via CPRX; all interaction with the program being debugged occurs on the target machine in the normal fashion.

Terminating CPRX and CPRK

If communications have not yet been established, you can terminate CPRX or CPRK by entering **CTRL-C** in the Shell window that you

used to invoke the command. If CPRX does not open its windows after you invoke it, communications have not been successfully established.

Once the program is loaded, you can choose to terminate debugging one of two ways:

- Terminate both CPRX and CPRK by entering the **finish** command in the CPRX Dialog window. The syntax is as follows:

```
fin[ish]
```

The program being debugged will automatically be terminated if it has not yet completed.

- Terminate only CPRX by using the normal CPR **quit** command. CPRK will continue running for use with a future CPRX debugging session. The program being debugged will automatically be terminated if it has not yet completed.

You should allow your program to run to completion if possible before entering **finish** or **quit** to ensure that it cleans up all resources.

6 Using AREXX Macros with CodeProbe

- 101 *The AREXX Interface*
- 101 *Macros Provided by CodeProbe*
- 102 *Invoking Macros*
- 103 *Values Returned from CodeProbe Commands*
 - 103 *Return Codes*
 - 103 *Command Output*

The AREXX Interface

AREXX is a multitasking implementation of the REXX language, a high-level language designed for macro processing and general programming tasks. CodeProbe supports an AREXX interface for customization. This interface is provided through CodeProbe's REXX port, `SC_CPR`.

Macros Provided by CodeProbe

A number of macros are provided in the `sc:rexx` directory as examples. Some simulate commands supported by the Commodore debugger, wack, including the following:

Macro	Description
<code>avail.cpr</code>	displays the amount of available memory
<code>dbptr.cpr</code>	dumps memory as bytes from a given BPTR
<code>dbstr.cpr</code>	dumps memory as bytes for a given BSTR
<code>devices.cpr</code>	displays the devices in the system
<code>devs.cpr</code>	displays the list of system devices
<code>execbase.cpr</code>	dumps exec base information
<code>ints.cpr</code>	displays the list of interrupt handlers

(continued)

Macro	Description
<code>libraries.cpr</code>	displays the libraries in the system
<code>libs.cpr</code>	displays the list of resident libraries
<code>makeaptr.cpr</code>	converts a BPTR to an APTR
<code>memory.cpr</code>	displays regions of memory
<code>mods.cpr</code>	displays system resident modules
<code>ports.cpr</code>	displays the list of public ports
<code>regions.cpr</code>	displays regions of memory
<code>resource.cpr</code>	displays the list of system resources
<code>rsrscs.cpr</code>	displays the resources in the system
<code>showcli</code>	dumps CLI structure for a given command, process address, or task
<code>showprocess.cpr</code>	takes a process address or name and displays all of its task and process fields
<code>status.cpr</code>	shows all CLI tasks
<code>whichis.cpr</code>	identifies the CLI process associated with a command, task number, or task address

For more details on how to use these macros while debugging, examine the macros in the text editor. Each macro has comments at the beginning of the file explaining how to use it.

Invoking Macros

CodeProbe recognizes macros with a `.cpr` extension. To invoke a macro from CodeProbe, simply enter the name of the macro from the command line. (The `.cpr` extension is optional.) CodeProbe first looks for the macro in the current directory, then in `sc:rexx`, and finally in `REXX:`.

For example, the following macro, named `avail.cpr`, uses the `storage` function to display the amount of memory available.

```
/* */
'd " || storage() 'bytes free"
exit(0)
```

To invoke this macro, enter **avail** from the command line. The **display** command in the **avail.cpr** macro displays the value returned by the **storage** function, 1953944, and the string “**bytes free**”, as shown here.

```
> avail
1953944 bytes free
```

Values Returned from CodeProbe Commands

CodeProbe commands provide return codes that are available to AREXX macros. Output from the debugger commands also can be assigned to the AREXX variable **results**.

Return Codes CodeProbe commands return status codes to AREXX as follows:

Return Code	Explanation
0	The debugger command was successful.
1	The command was syntactically correct, but the program failed for some other reason.
2	A syntax error occurred.
3	The debugger could not allocate memory for the AREXX message.

Command Output In addition to these return codes, an AREXX macro can receive the output of a debugger command as it would normally appear in the Dialog window by invoking the AREXX command **options results**. The output will then be accessible from the variable **result**, provided that the debugger returned a 0 return code.

For example, to execute a debugger command from outside the debugger, you could execute the following REXX script:

```
/* */
address "SC_CPR" /* only if macro was not invoked from CodeProbe */
options results
'any CodeProbe Dialog window command'
```

The output from the `CodeProbe` command is assigned to the AREXX variable **result**.

7 Debugging Resident Libraries

105	<i>Introduction</i>
105	<i>Setting Breakpoints</i>
106	<i>The symload Command</i>
106	<i>The libraries.cpr AREXX Macro</i>
106	<i>Cautions</i>
106	<i>Limitations</i>
106	<i>Example</i>

Introduction

Debugging resident libraries has been simplified considerably with Version 6.0 of CodeProbe. When debugging a resident library you can

- set breakpoints in the library
- use the **symload** command to read symbol information from the library
- use the **libraries.cpr** AREXX macro to display the libraries in the system.

Setting Breakpoints

You can set breakpoints in any resident library that your program may use. For example, the following command sets a breakpoint on the function **myfunc** in library **mylib.library**:

```
break mylib.library:myfunc
```

If your program has not yet called **OpenLibrary()** to open **mylib.library**, then CodeProbe waits to install the breakpoint until the library has been opened. When CodeProbe detects that the library has been opened, it loads the debug information for that library. If **myfunc** does not exist, or if debug information for that library cannot be found, CodeProbe will halt the program at that point. This allows you to specify a different function or use the **symload** command to load debug information from a different location.

The `symload` Command

The `symload` command can be used to load debugging information for a resident library. For example, the following command loads debugging information for the library named `mylib.library` in the current directory:

```
symload mylib.library
```

Refer to Chapter 9, “Commands and Built-in Functions,” for a complete description of the `symload` command.

The `libraries.cpr` AREXX Macro

The `libraries.cpr` AREXX macro can be used to display a list of the libraries that are currently loaded. To run this macro, enter `libraries` on the command line of the Dialog window. Refer to Chapter 6, “Using AREXX Macros with CodeProbe,” for more information.

Cautions

Since there is only one code segment for a shared library, you must be careful in setting breakpoints. A breakpoint is just an illegal instruction that CodeProbe places in the code. When the child process encounters that instruction, CodeProbe handles the exception. However, if another process is also using the same library, it will cause a crash when the program encounters the illegal instruction. Therefore, do not set breakpoints in libraries that will be used by processes not under debugger control.

Limitations

When debugging a resident library, you cannot examine global variables declared in modules outside the library even if you specify the full pathname of the module.

Example

See the directory `sc:examples/reslib` for an example resident library with debugging instructions.

8 Debugging Multitasking Programs

107	<i>Introduction</i>
108	<i>Types of Tasks</i>
108	<i>How CodeProbe Handles Tasks</i>
109	<i>Commands Used to Control Tasks</i>
109	<i>tasks</i>
110	<i>opt task</i>
111	<i>activate and deactivate</i>
111	<i>detach</i>
112	<i>catch</i>
112	<i>symload</i>
113	<i>Design Considerations for Debugger Compatibility</i>

Introduction

CodeProbe can debug applications that spawn additional tasks. These applications are designed to take advantage of the Amiga system's multitasking capabilities. This chapter looks at the commands used to debug multitasking applications including the following:

tasks

displays all tasks under the debugger's control.

opt task

examines or modifies the state of some task besides the one currently at a breakpoint.

deactivate

prevents a task from running, even when a **go** command is executed from another current task.

activate

reactivates a task affected by the **deactivate** command.

detach

frees a task from debugger control.

catch

places a task under debugger control.

symload

changes the load module used for collecting debug information while the debugger is active.

The discussion concludes by focusing on how to design applications with debugger compatibility in mind.

CodeProbe is able to debug multitasking applications by intercepting every call to `AddTask()` and determining the identity of the calling task. If the calling task is under debugger control and the `catch` option is set to `on`, the new task will also be brought under debugger control. Any task under CodeProbe's control can be stopped by a set breakpoint or by single stepping. Also, if you enter `Ctrl-C` in the Dialog window while an application is running, all active tasks under debugger control will be stopped.

Types of Tasks

Two types of tasks can be spawned by an application. The first type, called a *synchronous* task, is created, runs to completion, and returns control to the parent process while the parent process waits. This type of task is not practical to debug with CodeProbe while running the parent process. You should debug the task as a separate program before debugging the parent process.

The second type of task is an *asynchronous* task. When a parent process spawns an asynchronous task, control is returned to the parent process immediately after creation. The two applications then run independently, but both are under debugger control. This type of task can be debugged with CodeProbe.

In this chapter, all references are to asynchronous tasks and their parent processes.

How CodeProbe Handles Tasks

It is important to understand the way tasks are handled when a breakpoint is hit or when you are stepping through a task. If several tasks are running under the debugger's control when a breakpoint is reached, all other tasks on the debugger's list are stopped as well as the one that encountered the breakpoint. This guarantees that the state of the application remains consistent while the user examines the task. The debugger will set the environment to that of the breakpoint task. That is, the current module, current line, and the registers displayed will all belong to the breakpointed task. The name and address of the Task Control Block for the current task are always displayed in the Dialog window's title bar.

Commands Used to Control Tasks

CodeProbe provides a number of commands that are used to control tasks.

tasks The **tasks** command displays all tasks under the debugger's control:

```
>tasks
```

Address	Type	Pri	State	SigWait	StackPtr	Debug	Name
00C513B8	13	0	Waiting	80000000	00C551CC	act	multi

Note that there are eight fields (columns) displayed by this command:

Address

contains the address of the task control block.

Type

indicates the type of node that begins the task control block. The number 13 is used for processes, while 1 is used for simple tasks. For a complete list of the possible task types, see the header file **exec/nodes.h**.

Pri

contains the priority of the task.

State

indicates the process state.

SigWait

displays the **SigWait** field of the task control block. This field indicates which signal bits the task may be waiting on.

StackPtr

indicates the current position of the stack pointer. Note that this will be somewhat different from the value displayed in the **sp** register. The value displayed here includes any information placed on the stack by the trap handler or exception handler associated with this task.

Debug

indicates the task's status with the debugger: **act** denotes an active task, and **inact** denotes an inactive task. Tasks not under the control of the debugger will have no information in this field.

Name

refers to the name of the task as specified in the task's node structure.

For more information on tasks and any of the fields except the debug field just discussed, see the *Amiga ROM Kernel Reference Manual: Libraries and Devices*.

The **tasks** command accepts an option called **a11**. This option causes all tasks in the system to be displayed. This is useful if you want to catch a task that is not currently under debugger control. For more details, see the discussion on the **catch** command later in this chapter.

opt task The **opt task** command makes a new task current. This command is useful if several tasks are currently running under the debugger and you want to examine or modify the state of some task besides the one currently at a breakpoint.

When a new task is made current by this command, a new set of registers is displayed. Highlighting of changed registers in the Register window may be misleading since the debugger only keeps track of changes while stepping through a single task. The module displayed in the Source window may change, or more likely, if the task was in a resident or linked library, assembler lines will be displayed. It may be possible to get information as to the calling sequence of the task by using the **where** command, but since assembler routines, including the Amiga system's resident libraries, do not follow C language calling conventions, the information displayed by **where** is only an intelligent guess.

You can modify any registers, condition control register (CCR) flags, or stack variables, just as in the breakpointed task. You can even single step through the code. However, a word of caution is advised. A breakpoint is implemented by placing an illegal instruction at the desired location. All tasks under the debugger's control have a trap handler attached to catch the illegal instruction and report back to the debugger. A breakpoint in any shared code will insert an illegal instruction in any task that executes that code. If you must place a breakpoint in shared code, such as a resident library under test, be sure that any task that might open it is under debugger control. Never place breakpoints in libraries that you do not control. Improper use of breakpoints in these circumstances can crash your machine.

The **opt task** command, like all commands that manipulate tasks, takes *task-ID* as an argument:

```
op[t] t[ask] task-ID
```

A *task-ID* can be either the name of the task or the address of the task control block. If the name is used, it should be unique (otherwise

the first task with that name in the debugger's task list will always be selected). If the name contains a blank, the entire name must be surrounded by double quotes.

Task addresses are generally entered in hexadecimal with a `0x` prefix. Task names and addresses can be determined with the `tasks` command. Some sample commands are

```
>opt task Child
>opt task 0xC85428
>opt task "Child of multi"
```

Note: `task` command names are case sensitive.

activate and deactivate

While debugging a multiple task application, you may want to stop one or more tasks temporarily. The deactivate command allows you to prevent a task from running, even when the `go` command is performed from another current task. Later, the task can be reactivated by using the `activate` command. Both commands take *task-ID* as an argument.

```
a[ctivate] task-ID
dea[ctivate] task-ID
```

detach

At times, you may want to allow one or more tasks spawned by a task under the debugger to run freely and not under the control of CodeProbe. For example, a task designed to respond to Intuition events may cause the system to lock up if it does not respond to menu events quickly. It may not be desirable to allow such a task to be stopped when other tasks hit a breakpoint. Such a task can easily be freed from debugger control by using the `detach` command.

To use the detach command, first set a breakpoint at the entry point to the task. When the breakpoint is reached, use the `tasks` command to identify the address or name of the task (you may know this based on your code). Then enter the `detach` command. The syntax is the same as the other task manipulation commands previously discussed.

```
det[ach] task-ID
```

A few cautionary words are in order regarding the use of the `detach` command. Once the task is free from the debugger, it can no longer handle breakpoints. Therefore, be sure that all breakpoints in code reachable from a detached task have been removed. Second, the task will not be removed automatically if you quit the debugger. The

detached task can continue running even after you exit CodeProbe. Since CodeProbe frees all of the memory containing the code used by the detached task, subsequent code segments can use part of this free memory and a crash can occur.

If you later want to re-attach a task to the debugger, this can be achieved using the `catch` command, the subject of the following discussion.

catch At times, you may find that a process or task not started under the debugger is behaving in an unpredictable or undesired manner. For instance, the task may be caught in an infinite loop. Another possibility is that the task is waiting on some message port for a message that will never come. The `catch` command is used to capture tasks that are causing these types of problems. Its syntax is similar to those for other task commands:

```
ca[tch] task-ID
```

To debug a task that is stuck in an infinite loop, invoke CodeProbe with no filename. Then, when the debugger is up, use the command `tasks all` to find the name and address of the process in question. The `catch` command can then be used to capture the desired task and place it under debugger control. If the task is a process, this command will locate the code segments associated with the debug executable as available from the process structure. If the task is not a process, you will have to determine the address of the `seglist` and use the `opt env` command to tell the debugger how to map the debug information with the actual executable file. The `opt env` command takes as an argument the address of a `seglist` pointer, as in this example:

```
opt env 0xC08418
```

For more information on segment lists, see *The AmigaDOS Manual, 3rd Edition*.

symload The `symload` command enables you to change the executable file used for collecting debug information while the debugger is active. This is useful when debugging several interacting applications that are invoked from separate executable files. Note that this command does not load any code into memory or spawn any new processes or tasks.

The `symload` command can take several forms. The following command will find the executable file and the loaded memory segments associated with the specified process, if it has CLI structure:

```
symload proc "process-name" executable
```

The process is specified by the *process-name* parameter, and the executable file is specified by the *executable* parameter. Note that `symload` and `proc` are required keywords in this example. If the command has an executable filename as an additional argument, as in this example, any debug information associated with that file will be loaded instead of looking for the file associated with the process. The address of a process task block can be specified in place of the process name.

This command also can be given a pointer to a segment list as an argument:

```
symload seg 0x123456 executable
```

The keyword `seg` is followed by the hexadecimal address pointing to the process `seglst`. The name of an executable file is also required.

Design Considerations for Debugger Compatibility

CodeProbe makes every effort to allow an application to run as it would while running outside of debugger control. However, in order to establish breakpoints and catch new tasks generated in a multitasking application, CodeProbe must manipulate certain resources in the application's task structure. In particular, CodeProbe redefines a task's exception handler, exception data, trap handler, and trap data fields. If it is necessary for an application to redefine any of these fields for its own purposes, it must do so in the way described in the *Amiga ROM Kernel Reference Manual: Libraries and Devices*.

Exception handlers must save the address of the original exception handler and exception data fields and pass any unallocated exceptions back to those routines. Before invoking the original exception handler, be sure to restore the original exception data pointer in the task structure.

The same practice applies to trap handlers. In particular, all exceptions besides allocated traps should be passed to the original trap

handler. In order to control multitasking, CodeProbe performs a **SetFunction** on the **AddTask**, **OpenLibrary**, **OpenDevice**, and **RemTask** functions. These functions should not be redefined by an application.

9 Commands and Built-in Functions

115	<i>Introduction</i>
115	<i>CodeProbe Commands</i>
118	<i>Special Parameters</i>
118	<i>address</i>
119	<i>array-slice</i>
120	<i>expression</i>
120	<i>location</i>
121	<i>number</i>
121	<i>range</i>
122	<i>register</i>
123	<i>string</i>
123	<i>subrange</i>
124	<i>type</i>
125	<i>variable</i>
126	<i>Command Reference</i>
202	<i>Built-in Functions</i>
203	<i>Built-in Function Reference</i>

Introduction

This chapter provides complete reference information for each of the CodeProbe commands and the built-in functions. This information is provided in the following order:

- list of CodeProbe commands that shows acceptable abbreviations and provides a brief description of each command
- list of built-in functions with brief descriptions
- descriptions of special parameters, each of which is used with several CodeProbe commands
- reference information for each of the commands and built-in functions.

CodeProbe Commands

As explained in Chapter 1, “Introduction to CodeProbe,” debugger commands can be entered four ways: from the Dialog window, from a pull-down menu, by function keys, or by menu-accelerator keys. This

chapter describes each of the following debugger commands as they are entered from the Dialog window:

a[ctivate]	activates a task under debugger control
al[ias]	defines an alias for a debugger command
ar[gs]	displays the arguments to a function
bc[lear]	clears one or more breakpoints
bd[isable]	disables one or more breakpoints
be[nable]	enables one or more breakpoints
bl[ist]	lists all breakpoints
b[reak]	sets a breakpoint
c[all]	evaluates an expression and discards the result
ca[tch]	catches a task not currently under debugger control
dea[ctivate]	deactivates a task under debugger control
def[ine]	defines a macro
det[ach]	detaches a task from debugger control
d[isplay]	displays the value of an expression
du[mp]	dumps memory contents
dz[ero]	displays memory as a null-terminated ASCII string
ec[ho]	displays a string
env	sets the environment
ex[ecute]	executes a debugger command file
expand	expands and displays a command line
fin[ish]	terminates CPRK and CPRX
fr[egister]	displays or modifies floating-point registers
g[o]	continues execution until a breakpoint is encountered or the program exits
h[elp]	displays help information
hu[nks]	lists the addresses and sizes of all hunks
j[ump]	changes the current execution point

l[ist]	lists the lines in a source file
l[og]	logs debugger commands to a file
o[p]t	shows option values or changes the value of a debugger option
p[roceed]	single-steps over function calls
ps	single-steps over function calls by source line
q[uit]	terminates the debugger
r[egister]	displays or modifies registers
res[tart]	restarts the program being debugged
ret[urn]	returns immediately from the current function
rf[lag]	displays or modifies flags
sea[rch]	searches for a string in the current source file
se[t]	modifies the values of variables or memory locations
sle[ep]	pauses for the time specified
so[urce]	displays the source file
sta[rt]	restarts the program being debugged
symb[ol]	finds the symbol nearest to the specified address
sym[load]	reads symbol information from an executable file
ta[sks]	displays system tasks
t[race]	single-steps into function calls
ts	single-steps by source line into function calls
unal[ias]	deletes an alias
u[nassemble]	displays memory as assembler instructions
und[efine]	deletes a macro definition
w[at]ch	sets a watch on a variable or memory
wb[reak]	sets a watch break
wc[lear]	clears one or more watches
wd[isable]	disables one or more watches
we[nable]	enables one or more watches

<code>wha[tis]</code>	determines the type of an object
<code>wh[re]</code>	shows the calling sequence
<code>wi[ndow]</code>	opens or closes a window
<code>wl[ist]</code>	lists all watches
<code>wmsg</code>	writes a message to the Message window.

Special Parameters

The following special parameters are described in this section:

- ☐ *address*
- ☐ *array-slice*
- ☐ *expression*
- ☐ *location*
- ☐ *number*
- ☐ *range*
- ☐ *register*
- ☐ *string*
- ☐ *subrange*
- ☐ *type*
- ☐ *variable*.

Each of these parameters is used in several CodeProbe commands.

address An *address* parameter is any expression that denotes an address. This may be a hexadecimal constant, a C pointer variable, a register, the result of prefixing `&` to a C identifier of the proper type, or the result of arithmetic calculations on other addresses. Note that the address operator `&` may not be applied to the identifier of a bitfield or a register.

The following are examples of values of expressions that denote an *address* where `p` is a pointer, and `i` and `x` are integers:

```
&i
(&i + 8)
&array[3]
&mystruct - x
p
0x00C85400
a0
(a0 + 0x22)
```

There are some cases where a register name may be ambiguous. For example, you may have defined a variable `sp` in your program. In the following command it is not clear whether you mean to dump data beginning at the variable `sp`, or at the address contained in the register `SP`:

```
db sp
```

In such cases the debugger will assume that you mean to refer to the variable `sp`. However, you can prefix the register name with a dollar sign (\$) or use the `register` command to examine the value of the `SP` register even if you have declared a variable named `sp`.

In certain commands, it is necessary to specify the type of an object to be modified or displayed. If the object is referred to by means of an address constant or register, the value is assumed to be a character pointer (`char *`).

array-slice An *array-slice* parameter is a contiguous section of a unidimensional array of scalar variables, an ordered selection of elements from a multidimensional array, or an ordered selection of members from an array of complex types. An *array-slice* is specified using the following form:

```
variable [[n . . m ] | [* ]]
```

Note: The inner brackets in this syntax diagram are not optional and do not denote optional parameters.

The advantage of using an *array-slice* parameter over a *range* parameter is that the *array-slice* can be used to select isolated subcomponents, such as a particular structure member from an array of structures. The *array-slice* parameter also can be nested for multidimensional arrays or structure members that are arrays themselves.

The *subrange* parameter is described later in this chapter. If a *subrange* is chosen as the index of an *array-slice*, those elements within the *subrange* are selected. The asterisk (*) index specifies that all elements of the array are to be contained in the *array-slice*.

The following are examples of the *array-slice* parameter:

```
a[3..7]
matrix[i..j][1..3]
b[*]
a[3..6]->b
```

expression Any C expression is accepted as an *expression* parameter, except those that use one or more of the following operators:

```
++ or --
,
?:
=
&=
|=
>>=
<<=
+=
-=
*=
/=
%=
^=
```

location A *location* parameter specifies a place in the code at which a breakpoint is to be set, code is to be unassembled, or a similar action is to be performed. When debugging in source mode, you can think of a *location* parameter as a line number in a source file or a function entry point. When debugging in assembly mode, you may want to place breakpoints at specific addresses. To do this, specify a hexadecimal address as the *location* parameter.

The *location* parameter can be specified in one of two ways:

- *hex-address*
- `[[image : ][\ module \] function ] line`

In the first form, the *hex-address* variable can be any valid absolute address specified as a hexadecimal integer.

In the second form the values for the variables *image*, *module*, and *function* can be specified explicitly as follows:

image

is the name of the executable image (the program, library, or device) containing the location. It is usually specified in lowercase. If omitted, the current image is the default. In a program with only one image, it is never necessary to specify a value for *image*: Note the required colon (:).

module

is the name of the C source file compiled to yield the specified function. Note the backslash (\) postfix.

function

is the name of a function in the application.

You can put spaces between the value specified for *image* and its colon postfix and the value specified for *module* and its backslash postfix, but they are not necessary. You must put a space before the *line* parameter.

The *line* parameter can specify any of the following points in the source code:

integer

a line number, relative to the start of the C source file containing the function

\$

the current location

e[ntry]

the entry to the function (prolog)

r[eturn]

the return from the function (epilog).

If the function name is not specified with the *line* parameter, the debugger defaults to the current function. If a function is specified without a line, the debugger defaults to the first execution line.

number A *number* parameter is a number in decimal, octal, or hexadecimal notation. An initial 0 digit causes the number to be interpreted as octal, an initial 0x (or 0X) causes the number to be interpreted as hexadecimal, and an initial 0n (or 0N) causes the number to be interpreted as decimal. The default is decimal.

Examples of acceptable values for the *number* parameter include the following:

```
12345
0455
0x380
0X1849
0n12345
```

range A *range* parameter is a contiguous area of memory that can be specified in one of two ways:

- ☐ address . . address
- ☐ address l number
address L number

In both forms, the first *address* variable is the start address, and the range begins at this address.

In the first form, the second value for *address* is the end address, and the range includes the area of memory from the start address to the end address.

In the second form, the **1** or **L** indicates that the value for the variable *number* is to be interpreted as a length. In this case, *number* indicates the number of elements and the range is from the start address through $\text{address} + \text{number} - 1$. When the debugger sees a solitary **1** or **L** in a command, it is considered part of a *range* expression.

The following examples fit the definition of the *range* parameter:

```
0x123456 .. 0x123461
0x123456 L 11
```

```
a1 .. a2
a7 1 20
```

```
&p[0] .. &p[5]
p[0] .. p[5]
```

```
&p[0] 1 6
p[0] 1 6
```

The first pair of examples are equivalent. The next pair are variants of the basic *range* expression. The remaining pairs are equivalent ranges, given the addressing policy described earlier in this section.

register A *register* parameter refers to the name of any of the following 680x0 registers:

- A0 - A7 are address registers. Register A7 functions as the stack pointer and can be specified as either A7 or SP.
- D0 - D7 are general-purpose registers.
- PC is the program counter.
- SP is the stack pointer.
- SR is the status register. It contains the CCR register as well as the current processor status.
- CCR is the condition code register. It resides in the lower byte of the status register, SR.

If a math coprocessor is present, the following registers also can be specified as the *register* parameter:

FP0 - FP7 are the floating-point registers.

FCR is the floating-point control register.

FSR is the floating-point status register. It contains the current floating-point condition codes.

Optionally, registers can be prefixed with a dollar sign (\$) to distinguish them from variables with the same name. For example, register D1 can be specified as \$D1 to distinguish it from a variable named `d1`.

string A *string* parameter can be any standard C string in double quotes ("). Standard C escape sequences starting with the backslash character are supported as follows:

<code>\n</code>	newline (0x0a)
<code>\t</code>	horizontal tab (0x09)
<code>\b</code>	backspace (0x08)
<code>\r</code>	carriage return (0x0d)
<code>\f</code>	form feed (0x0c)
<code>\v</code>	vertical tab (0x0b)
<code>\NNN</code>	octal constant, where NNN is a three-digit octal number
<code>\xNN</code>	hex constant, where NN is a two-digit hexadecimal number.

A backslash followed by any character other than those shown previously in the escape sequences is interpreted as a plain character. For example, `\\` is a string consisting of a single backslash, and `a\"b` is three characters long: an `a`, a double quote, and a `b`.

The Dialog window does not recognize any of these escape sequences when they are displayed. You cannot use the `\t` escape sequence to tab in the Dialog window. The escape sequences are useful in line mode and when setting a character string.

Like all debugger input, a string can be continued to the next line by ending the line with a backslash. The debugger does not support ANSI string concatenation.

subrange A *subrange* parameter is a series of contiguous integers, inclusive of its bounds. The integers may be either integer constants or variables from

the application being debugged. The form of the *subrange* parameter is as follows:

`[integer | variable] .. [integer | variable]`

The following are examples of the *subrange* parameter:

`3..7`

`i..j`

type A *type* parameter can be any of the data types provided by the C language including any of the following:

```
char
unsigned char
short
unsigned short
int
unsigned int
long
unsigned long
unsigned
float
double
struct <name>
union <name>
enum <name>
```

A *type* also can be any of the C types previously listed followed by some number of asterisks indicating that the type is a pointer to the base object.

In addition, the *type* parameter can take any identifier defined by means of a **typedef** statement if the debugging information for the module supplies typedef information.

variable A *variable* parameter is an expression in your program that designates an object. It can be either a simple identifier or an expression referring to an array element, structure member, or object pointed to by a pointer. In addition, it can begin with a specification of the following form:

```
[[image:][\module\]function]
```

See the *location* parameter earlier in this chapter for a description of the *image*, *module*, and *function* variables.

The following are objects that fit the definition of the *variable* parameter:

```
i
max
array[3]
array
io:readfile\length
mystruct->name[2]
mystruct
*cptr
main.c\opnf\count
opnf\i
mylib.library:LIBmyfunc\myvar
```

Command Reference

This section provides complete reference information for each of the CodeProbe commands. The commands are listed in alphabetical order. Most of the command descriptions include the following sections:

<i>Synopsis</i>	shows the format of the debugger command or built-in function
<i>Description</i>	describes the function of the debugger command or built-in function and contains all the information you need to use the command or function
<i>Example</i>	contains examples that illustrate how to use the command or function
<i>See Also</i>	refers you to the descriptions of other related commands or functions.

activate Activates a task under debugger control

Synopsis `a[ctivate] [task-name | task-address | exe-name]`

Description The **activate** command activates a task that has been deactivated by the **deactivate** command. The task is activated when the next **go** or **proceed** command is executed.

The *task-name*, *task-address*, and *exe-name* parameters are the name of the task, the address of the task, and the executable name of the task as specified by the **tasks** command. Only one of these parameters can be specified.

Examples

```
activate Child
    activates the task named Child under debugger control.

activate 0xC08540
    activates the task with a task block starting at address 0xC08540.

activate myprog
    activates the task executing the program myprog.
```

See Also `catch`, `deactivate`, `detach`, `tasks`

alias Defines an alias for a debugger command

Synopsis **al[ias]** [*name*[*definition*]]

Description When entered with no arguments, the **alias** command displays the list of all currently defined aliases. If a value for *name* is provided as the only argument, the definition for that alias will be displayed. Any text following the *name* parameter is treated as a *definition* and is used to define the alias.

Alias names are expanded only when they occur at the beginning of a Dialog window command line. An alias definition can be a simple textual substitution or it can contain positional parameters to be expanded at execution time. Positional parameters are specified as **\$1**, **\$2**, **\$3**, and so on. **\$0** represents the alias name itself. **\$*** represents command line parameters **\$1** through **\$n**. If no positional parameter is specified in a definition, all parameters are placed after the expanded definition text.

The **alias** command will parse a definition until it encounters a command delimiter character (;) or the end of the line. To define an alias that expands into more than one command, enclose the definition in either double quotes or curly braces. If the command string must contain double quotes, curly braces can be used or the quotes can be escaped. As in C and all other debugger commands, a backslash may be used to continue the line.

Examples **alias**

displays a list of all alias definitions.

alias foo

displays a definition of **foo**.

alias next proceed

defines **next** to be an alias for **proceed**.

alias pr display foo, bar, \$*

defines **pr** to be an alias that executes the **display** command printing the value of **foo**, **bar**, and any variables entered after the **pr**.

alias doit {go \$1; d foo}

defines **doit** to execute the **go** command using the first parameter, and displays the variable named **foo**.

**alias doit {go \$1;\
d foo}**

uses the backslash command to split a command across two lines.

alias Defines an alias for a debugger command
(continued)

```
alias doit "go $1; d $2"
```

uses positional parameters with multiple commands. The first parameter, **\$1**, is passed to the **go** command, and the second parameter, **\$2**, is passed to the **display** command.

See Also **define**, **unalias**, **undefine**

args Displays the arguments to a function

Synopsis **ar[gs]** [*function-name*]

Description The **args** command displays all of the argument names and values to the current function or the function specified by the *function-name* parameter. If a *function-name* parameter is specified it must be in the calling sequence, as displayed by the **where** command.

Examples **args**
displays the arguments to the current function.
args sort
displays the arguments to the **sort** function.

See Also **display, env, where**

bclear Clears one or more breakpoints

Synopsis `bc[lear] integer[integer...]
 bc[lear] * | l[ast]
 bc[lear] integer .. integer`

Description The **bclear** command clears one or more breakpoints. When a breakpoint is cleared, it ceases to exist and can be reinstated only by issuing the **break** command again. You should use the **bdisable** command to disable breakpoints temporarily.

The *integer* parameter specifies the breakpoint number as displayed by the **blist** command. Any number of values can be specified as **integer** parameters. An asterisk specifies that all breakpoints should be cleared and **last** causes the most recently set breakpoint to be cleared. A range of breakpoints can be specified by *integer .. integer*.

Examples `bclear 2 5 6`
 clears the breakpoints numbered 2, 5, and 6.

`bclear last`
 clears the last breakpoint set.

`bclear *`
 clears all breakpoints.

`bclear 4 .. 7`
 clears breakpoints 4, 5, 6, and 7

See Also **bdisable**, **benable**, **blist**, **break**, **wclear**

bdisable Disables one or more breakpoints

Synopsis **bd[isable]** *integer*[*integer* . . .]
bd[isable] * | **l[ast]**
bd[isable] *integer* .. *integer*

Description The **bdisable** command disables the recognition of one or more breakpoints. When a breakpoint is disabled, it is not recognized by CodeProbe, but it does remain on the list of current breakpoints. A disabled breakpoint can be re-enabled by the **benable** command.

The *integer* parameter specifies the breakpoint number as displayed by the **blist** command. Any number of values can be specified as *integer* parameters. An asterisk specifies that all breakpoints should be disabled and **last** causes the most recently set breakpoint to be disabled. A range of breakpoints can be specified by *integer* .. *integer*.

Examples **bdisable 2 5 6**
disables the breakpoints numbered 2, 5, and 6.

bdisable last
disables the last breakpoint set.

bdisable *
disables all breakpoints.

bdisable 4 .. 7
disables breakpoints 4, 5, 6, and 7.

See Also **bclear**, **benable**, **blist**, **break**, **wdisable**

benable Enables one or more breakpoints

Synopsis **be**[enable] *integer* [*integer* ...]
 be[enable] * | **l**[ast]
 be[enable] *integer* .. *integer*

Description The **benable** command enables the recognition of one or more breakpoints that have been disabled by the **bdisable** command. The *integer* parameter specifies the breakpoint number as displayed by the **blist** command. Any number of values can be specified as *integer* parameters. An asterisk specifies that all breakpoints should be enabled and **last** causes the most recently set breakpoint to be enabled. A range of breakpoints can be specified by *integer* .. *integer*.

Examples **benable 2 5 6**
 enables the breakpoints numbered 2, 5, and 6.
 benable last
 enables the last breakpoint set.
 benable *
 enables all breakpoints.
 benable 4 .. 7
 enables breakpoints 4, 5, 6, and 7.

See Also **bclear**, **bdisable**, **blist**, **break**, **wenable**

blist Lists all breakpoints

Synopsis **bl**[ist]

Description The **blist** command displays a list of all breakpoints. Each entry has a form similar to the following example:

```
5 0x40054 loop:\loop.c\main 8 (2 hits)
   after(4) when(i == 3) quiet {echo Hi}
```

The number 5 in this example is the number used to refer to this breakpoint in **bclear**, **bdisable**, and **benable** commands. If an asterisk appears next to the number, that breakpoint has been disabled and will not be triggered. Following the breakpoint number is the hexadecimal address at which the breakpoint resides. If sufficient debug information is available, the address is followed by the location that it represents. The hit count, which represents a running count of the number of times the breakpoint has been triggered, is displayed after the location of the breakpoint. A breakpoint is triggered when all of the following conditions are true:

- ☐ execution has reached the address shown
- ☐ the **when** and **after** conditions are true, if specified
- ☐ the breakpoint is enabled.

Breakpoints are triggered when the user steps to the address provided the last two conditions are met. The second line of output displayed by the **blist** command shows the options given when the **break** command was issued.

See Also **bclear**, **bdisable**, **benable**, **break**, **wlist**

break Sets a breakpoint

Synopsis **b[reak]** *location*[**a**fter(*integer*)] [**w**hen(*expression*)]
 [**t**r[ace]] [**q**uiet]] [**t**e[m]p]] [*{cmd-list}*]

Description The **break** command is used to set a breakpoint at a location specified by the *location* parameter. You can use a dollar sign to specify the current location.

The **a**fter option specifies the minimum number of times the specified line must be executed before the breakpoint is triggered. This is also known as specifying a pass count. For **break** commands that do not contain an **a**fter clause, the pass count is set to 1.

Each time the breakpoint is hit, the pass count is tested. If the pass count equals 1, the breakpoint is triggered and execution stops. If the pass count is greater than 1, the pass count is decremented by 1, and execution continues. The **b**list command shows the current pass count for each breakpoint.

If the **w**hen option is present, the breakpoint is triggered only if the specified *expression* evaluates to true (nonzero). The **a**fter and **w**hen options can be specified in any order. If both options are specified, the pass count is tested and decremented only when the **w**hen condition is true.

The **t**race option continues execution automatically after the breakpoint is triggered, unless you reach the breakpoint by single-stepping, then the **t**race option has no effect.

The **q**uiet option suppresses the default message showing the location when the breakpoint is triggered.

The **t**emp option causes the breakpoint to be deleted after it has been triggered.

If the *cmd-list* parameter is present, the commands listed are performed each time the breakpoint is triggered. As in C commands and all other debugger commands, a backslash may be used to continue the line.

Note: To avoid confusion, the **break** command does not use the default radix setting. You must specify **0x** for an address; otherwise, you will be specifying a line number. For example, even if the radix is set to **hex**, the following command specifies a **break** command for line 20 and not address **0x20**:

b 20

break Sets a breakpoint
(continued)

Examples **break \$**
sets a breakpoint at the current line.

b 14
sets a breakpoint at line 14 of the current module.

b sort
sets a breakpoint at the first line of the **sort** function.

break sort return
sets a breakpoint at the return from the **sort** function.

break \myfile.c\sort 14
sets a breakpoint at line 14 of **sort** in the module named **myfile.c**.

break mylib.library:myfunc
sets a breakpoint at the first line of the **myfunc** function in the shared library **mylib.library**.

b \$ after(5)
breaks the fifth time the current line is executed.

break 0x804A
sets a breakpoint at the absolute address **0x804A**.

**break 14 when (i > 5) {di p L 16; b sort 26
when (i == 8); bl; go}**
sets a breakpoint on line 14 that performs the commands given inside the curly braces if **i > 5** evaluates true.

b 8 trace quiet {d "i = ", i}
prints a message of the form "**i = value**" every time line 8 is executed.

break 100 temp {d foo}
stops at line 100, prints **foo**, and deletes this breakpoint.

See Also **bclear**, **bdisable**, **benable**, **blist**, **go**

call Evaluates an expression and discards the result

Synopsis `c[all] expression`

Description The **call** command is used to evaluate an *expression* when you do not care about the value returned. It is equivalent to the following statement in C language:

expression;

The most common use of the **call** command is to call a function in the application program, though function calls also may appear in arbitrary expressions in other commands. If you want to see the value returned by the expression, use the **display** command.

The debugger looks at the types of the parameters specified and not at the definition of the function or any prototype. Thus, the debugger does not flag an error if the wrong number of parameters are specified, nor does it perform automatic casting of types or allow passing of **char**, **short**, or **float** types. For example, the following command passes two parameters, an **int** and a **double**, regardless of how the function is declared:

```
call f(2, 1.5)
```

CodeProbe supports standard C syntax for function calls. Either a function name or an expression evaluating to a function type may be specified before the opening parenthesis. Thus, the **call** command can be used only to call functions or evaluate expressions that point to a function. For example, if **pf** is a function pointer, and **func** is a function, the following commands are allowed:

```
call func()
call (*pf)()
```

However, the following commands are not allowed:

```
call pf()
call 0x134()
call register()
```

You cannot use the **call** command to execute a function after a program has ended.

call Evaluates an expression and discards the result
(*continued*)

If a breakpoint is hit, a signal is caught, or the program terminates in the middle of a function call, the debugger aborts the expression and leaves you at that location. A warning is printed if any parameters are not cleaned off the stack. This may cause a return to an invalid location later if execution is allowed to continue.

Examples `call status()`
calls the `status` function.

`call print("Test", 300)`
calls the `print` function passing in a string and an integer parameter.

`call (*fp)(&arr[5], j+10, 3.0)`
evaluates arbitrary expressions.

See Also `display`, `env`, `where`

catch Catches a task not currently under debugger control

Synopsis `ca[tch] [task-name | address | exe-name]`

Description The **catch** command enables the debugger to gain control of tasks that were not started under the control of CPR. The *task-name* and *address* parameters can be found by issuing a **tasks all** command. The *exe-name* parameter can be used to specify the name of the executable associated with the task you want to catch.

Examples `catch Child`
 catches the task named **Child**.
`ca 0xC08540`
 catches the task with a task block starting address **0xC08540**.
`catch myprog`
 catches the task running the program **myprog**.

See Also **activate, deactivate, detach, symload, tasks**

deactivate Deactivates a task under debugger control

Synopsis **dea**[ctivate] [*task-name* | *address* | *exe-name*]

Description Normally all tasks under CPR's control are started when a **go** or **proceed** command is issued. The **deactivate** command can be used to deactivate a task. A task that is deactivated by this command is not started when the debugger allows the application to run unless it is the current task. The task can be reactivated with the **activate** command.

The *task-name* and *address* parameters can be found by issuing a **tasks** command. The *exe-name* parameter can be used to specify the name of the executable associated with the task you want to catch.

Examples **dea Child**

deactivates the task named **Child** under CPR.

deactivate 0xC08540

deactivates the task with a task block starting at address **0xC08540**.

deactivate myprog

deactivates the task running the program named **myprog**.

See Also **activate, catch, detach, tasks**

define Defines a macro

Synopsis `[#]def[ine] name [definition]`
`[#]def[ine] name(parm [,parm ... ]) [definition]`

Description The **define** command provides a general macro substitution facility that is nearly identical to the mechanism provided in C syntax. Macros defined in the debugger are expanded anywhere within a line of text except inside quotes or filenames. The debugger also will not expand macro definitions that are prefixed with a backtick character (`). The ANSI # and ## operators are not supported.

Examples **define**
displays all the macro definitions.

define foo
defines **foo** as a null definition.

def foo bar
defines **foo** to be **bar**.

define func(a,b) (a+b)
defines **func** with parameters.

See Also **alias, unalias, undefine**

detach Detaches a task from debugger control

Synopsis **det[ach] [task-name | address | exe-name]**

Description The **detach** command detaches a task from debugger control. The task can be brought under CodeProbe's control with the **catch** command.

The *task-name* and *address* parameters can be found by issuing a **tasks** command. The *exe-name* parameter can be used to specify the name of the executable associated with the task you want to catch.

When using the **detach** command make sure that no breakpoints are placed inside code that is executable from a detached task. Also, do not specify the **detach** command to the original program if any of its child processes continue to run in the same code segment.

Examples **detach Child**

detaches the task named **Child** from debugger control.

det 0xC08540

detaches the task with a task block starting at address **0xC08540**.

det myprog

detaches the task executing the program **myprog**.

See Also **activate, catch, deactivate, tasks**

display Displays the value of an expression

Synopsis `d[isplay] (expression["format"] | string) [, ...]`

Description The **display** command evaluates the value of the *expression* parameter and displays the result. The *expression* parameter can be any C expression containing constants, variables, and function calls from the program being debugged. Any number of expressions and strings can be specified, separated by commas. All expressions are displayed on the same line.

The **display** command chooses the default *format* based on the type of the expression. Values of type **char** are displayed as characters as well as in decimal and hexadecimal format. If the character is unprintable it is displayed as an escape sequence, for example, `\n` or `\xAC`. Values of type **int** and **long** are displayed in decimal and hexadecimal format, and **floats** and **doubles** are displayed in floating-point best format.

To override the default format, a **printf** style format can be specified optionally after each expression. No error checking is done on the *format* parameter, so strange results are possible. It may contain up to two conversion operators for **short** and **long** types but only one for all other types.

All members of structures, unions, and arrays are displayed using the default formats. Partial arrays may be displayed using two integers to indicate the first and last elements to display. For example, the following command displays array elements 3, 4, and 5:

```
display a[3..5]
```

An asterisk may be used instead to indicate that all elements should be displayed.

The *string* parameter can be any standard C string in double quotes as described earlier in this chapter in the “Special Parameters” section.

To display a null-terminated string, use the **dzero** command. Use the **dump** command to dump an area of memory.

Examples `display i`
displays the variable `i`.

`d a "a=%10.5e"`
displays `a` using a **printf** style format specifier.

`d p->d[5] + f(3)`
displays the value of an expression containing a function call.

display Displays the value of an expression
(continued)

```
display "X is", x, "J is", j
```

displays the value of several arguments including strings.

```
d x "X is %d", j "J is %d"
```

displays the value of several arguments using `printf` style format specifiers.

```
d a[3]
```

displays the value of an element of an array.

```
display a[3..5]
```

displays the value of elements 3 through 5 in an array named `a`.

```
d a[*]
```

displays the value of every element of an array.

See Also `call`, `dump`, `dzero`

dump Dumps memory contents

Synopsis `du[mp] [variable | address | range] [$]
[format [itemsizes] | text]`

Description The **dump** command is used to examine an area of memory. You select the format of the dump, the length of each item being dumped, and whether or not you want text to appear on the right side of the dump. A **dump** command with no parameters continues where the previous **dump** command ended using the same dump specifications.

The *variable*, *address*, or *range* parameter specifies the location to be dumped. The **\$** parameter can be used to specify the last dump address when the *format* or *text* parameters are used. The **\$** is ignored if a *variable*, *address*, or *range* parameter has already been specified making it useful for aliases where the dump address may be omitted.

The *format* parameter controls the format of the dump and its value can be any of the following:

a [scii]	FFP [float]	IEEE [float]
b [inary]	f [loat]	o [ctal]
d [ecimal]	h [ex]	u [nsigned]

The default format is **hex**.

If you specify a *format* parameter, you also can specify an *itemsizes* parameter to control the length of each item dumped. The *itemsizes* parameter can have a value of 1, 2, 4, or 8. The *itemsizes* parameter is concatenated onto the end of the *format* parameter (for example, **b4** for a binary format of length 4). If you do not specify *itemsizes*, CodeProbe uses a default. As shown in the following table, the allowable size and default values depend on the *format* specified.

The *text* parameter is used to specify that an ASCII representation of the memory dump should be displayed.

dump Dumps memory contents
(continued)

The *itemsize* parameter has the following default values and allowable sizes:

Format	Default	Allowed Sizes
ascii	1	1
binary	1	1, 2, 4
decimal	4	1, 2, 4
FFPfloat	4	4
float	8	4, 8
hex	4	1, 2, 4
IEEEfloat	8	4, 8
octal	4	1, 2, 4
unsigned	4	1, 2, 4

Compatibility Aliases

To provide backward compatibility with previous releases of CodeProbe, the following aliases for the **dump** command are supported:

- da**
dumps a range of memory as ASCII characters
- db**
dumps a range of memory in both hexadecimal and ASCII format
- dc**
dumps a range of memory in decimal format
- dd**
dumps a range of memory as 8-byte floating-point numbers in IEEE format
- df**
dumps a range of memory as 4-byte floating-point numbers in IEEE format
- df fp**
dumps a range of memory as 4-byte floating-point numbers in FFP format

dump Dumps memory contents
(continued)

di or **dl**

dumps a range of memory in integer format using decimal representations of the integers

dw

dumps a range of memory in short integer (16-bit) format using hexadecimal representations of the integers

dp

dumps a range of memory in 4-byte hexadecimal format

ds

dumps a range of memory in short integer (16-bit) format, using decimal representations of the integers.

Examples **dump var**

dumps memory contents of variable **var**.

dump var L 13 hex1

dumps memory contents for 13 bytes using 1-byte hexadecimal format.

dump var1 .. var2 ascii

dumps memory contents between **var1 .. var2** in ASCII format.

dump 0xAF8100 L 8

dumps memory contents for 8 longs (32 bytes) starting at absolute address **0xAF8100**.

See Also **display**, **dzero**

dzero Displays memory as a null-terminated ASCII string

Synopsis **dz[ero]** *variable* | *address* | *array-slice*

Description The **dzero** command displays the contents of memory as a null-terminated (`\0`) string. If the location is the name of a nonpointer *variable* in the program being debugged, **dzero** displays the memory contents starting at the address of the variable and continuing until the null character is found. If the location is the name of a pointer variable, **dzero** uses the contents of the pointer and displays bytes starting at that address until a null character is encountered. If the *address* parameter is used to specify an absolute address, **dzero** displays the contents of that address until a null character is encountered. If the location is an *array-slice*, the address of each element in the array slice will be displayed until the null character is found.

Examples **dzero string**
 displays characters until the null character is reached.

dzero ptr
 displays data pointed to by **ptr** as a string.

dzero ptr[3 .. 7]
 displays each element in the array slice **ptr[3 .. 7]** until the null character is reached.

See Also **display**, **dump**

echo Displays a string

Synopsis **ec[ho]** [*text*]

Description The **echo** command writes the value specified for the *text* parameter in the Dialog window. This is useful for documenting the actions being taken in a debugger command file, the **cprinit** file, or AREXX macros. If the *text* parameter contains semicolons, they must be escaped with a backslash (\;). Defines or aliases are not expanded.

Example **echo Starting program\; loading defines.**
displays the message following the **echo** command in the Display window.

See Also **display, execute**

env Sets the environment

Synopsis `env[function | integer]`
`env[-c[aller] | -s[subroutine] | -u[p] |`
`-d[own] | -integer | +integer]`

Description The **env** command changes the environment for subsequent debugger commands. *Environment* is the state of the machine at a particular point in the calling sequence (that is, the contents of the stack and the contents of variables). Setting the environment to a previous point in the calling sequence, such as the point where the current function was called, returns the machine to the state it was in at that time. However, only the information that can be retrieved from the stack will have been restored. Scratch registers and other components, such as the values of **externs**, that are not dependent on the stack remain relative to the current execution point in the application and will not be changed. Nonscratch registers are restored to the correct values for the new environment.

The environment can be set to an absolute or relative position in the call chain. To set it to an absolute position, specify the function name using the *function* parameter, or a level number using the *integer* parameter. The **where** command or Calls window can be used to display the level number. The other forms to the **env** command set the environment relative to the current function.

To move the environment up one level to the caller of the current function, use the **-caller**, **-up**, or **-1** flags. To move in the opposite direction, use the **-subroutine**, **-down**, or **+1** flags. Use the **+integer** or **-integer** forms to move more than one level at a time.

Level 1 is defined to be the function in which you are currently stopped. The caller's level is 2, its caller is level 3, and so on. These level number designations change as the program steps into and returns from functions.

If you attempt to move the environment up or down more levels than there are in the call frame, a warning is displayed and the environment is moved as far as possible.

After using the **env** command, commands that refer to line numbers and variables act as if the function specified by the *function* parameter is the current function. Any command that causes the program being debugged to resume execution, such as the **go** or **trace** commands, automatically resets the user environment to the last point of execution. Issuing the **env** command is the same as double-clicking on a function call entry in the Calls window.

env Sets the environment
(continued)

Examples **env**
 sets the user environment to the last point of execution.
env main; d i
 sets the user environment to **main** and display **i**.
env -subroutine
 sets the environment to the called function (down 1 level).
env -caller
 sets the environment to the caller function (up 1 level).
env +5
 moves the environment down 5 levels.
env 7
 moves the environment to the level 7.

See Also **where**

execute Executes a debugger command file

Synopsis **ex[ecute]** *filename*

Description The **execute** command reads and executes CodeProbe commands from the file specified by the *filename* parameter. If the file cannot be found, CodeProbe appends a **.cpr** extension to it and tries again. Thus, you can place a set of debugger commands in a file with a **.cpr** extension and simply use the root filename, minus the extension, in the **execute** command. CodeProbe searches for the specified file in your normal Shell path as defined with the AmigaDOS **PATH** command.

If the file **cmds.cpr** consists of the commands

```
break sort
blist
trace
trace
```

then the **execute** command applied to this file would yield the following display:

```
> execute cmds.cpr
executing commands from cmds.cpr
1 0xC32D3E sort:\sort.c\sort 22
sort:\sort.c\init 5
sort:\sort.c\init 9
```

The debugger commands themselves are not echoed as part of the display unless echo mode is turned on.

If you are debugging a program frequently and want to set a number of breakpoints at the same places, you can place all of your breakpoint commands in a file and then use the **execute** command on this file. This way you can avoid re-entering the same commands each time you want to use the debugger.

If the **execute** command is used in a command list, it must be the last command in the list.

Examples **execute setvars**
executes the file named **setvars** or **setvars.cpr**.

execute test/cmds.cpr
executes the file named **cmds.cpr** located in the **test** directory.

expand Expands and displays a command line

Synopsis **expand** [*alias*] *command-line*

Description The **expand** command displays its arguments with all aliases or defines expanded. The *command-line* parameter specifies any valid command-line entry that includes aliases or defines. This command can be useful in trying to create more complex aliases and defines.

If the **alias** keyword is not specified, aliases are not expanded, but defines will be expanded.

Examples **expand alias dp**
expands the alias for named **dp** to display the following:

```
dump                $ hex4
```

expand ISEQUAL(1,5)
expands the macro **ISEQUAL** using the arguments 1 and 5. For example, if **ISEQUAL** has been defined as (**a == b**), then it would be expanded to (**1 ==5**).

See Also **alias**, **define**

finish Terminates CPRK and CPRX

Synopsis **fin**[ish]

Description The **finish** command is used in cross-development mode to terminate the session. The cross debugger (CPRX) on the host machine instructs the kernel (CPRK) on the target machine to terminate after cleaning up and closing the communications link. The **finish** command must be entered on the host machine. The host machine is the one that is being used to display the CodeProbe user interface. The **finish** command does not work from the native version of CodeProbe.

Example **finish**
causes CPRK to terminate after disconnecting from CPRX.

See Also **quit**

fregister Displays or modifies floating-point registers

Synopsis `fr[egister] [register[|=] expression]`

Description On machines that have a math coprocessor chip installed, the **fregister** command displays or modifies the contents of the floating-point registers. If no parameters are specified, the **fregister** command displays the current contents of all the machine's floating-point registers in hexadecimal and as floating-point numbers. If only a register name is supplied, the contents of the register are displayed; otherwise, the value of the *expression* parameter is stored in the *register* specified.

The **display** and **set** commands also can be used to display and modify the registers.

Examples `fr`
displays all floating-point registers and flags.

`fregister fp0 30.14`
sets floating-point register `fp0` to `30.14`.

`fr fp2 = sales`
sets floating-point register `fp2` equal to the value of `sales`.

`fregister fp1`
displays the value of floating-point register `fp1`.

See Also `display`, `register`, `rflag`, `set`

go Continues execution until a breakpoint is encountered or the program exits

Synopsis `g[o][location[after(integer)] [when(expression)]]`

Description The **go** command begins execution of the program at the current location, which is identified by the address stored in the program counter (pc) register. Execution continues until a breakpoint is reached or until the program terminates, either normally or with an error.

The *location* parameter specifies an optional location for a temporary breakpoint. The breakpoint specified in the **go** command exists only for the duration of the **go** command. The next time program execution stops, the breakpoint is cleared, even if the temporary breakpoint was not reached.

The **after** option specifies the minimum number of times the specified location must be reached before the temporary breakpoint is triggered. This is also known as specifying a pass count. For **go** commands that do not contain an **after** clause, the pass count is set to 1.

Each time the temporary breakpoint is hit, the pass count is tested. If the pass count equals 1, the temporary breakpoint is triggered and execution stops. If the pass count is greater than 1, the pass count is decremented by 1 and execution continues.

If the **when** option is present, the breakpoint is triggered only if the specified *expression* evaluates to true (nonzero). The **after** and **when** options can be specified in any order. If both are used, the pass count is only tested and decremented when the condition is true.

Note: To avoid confusion, the **go** command does not use the default radix setting. You must specify **0x** for an address; otherwise, you will be specifying a line number. For example, even if the radix is set to **hex**, the following command specifies a **go** command for line 20 and not address **0x20**:

```
g 20
```

Examples **go**

continues program execution to the next breakpoint or the end of the program.

```
go 14
```

continues program execution to line 14 of the current module. Note that line numbers are relative to a module and not a function. This example assumes that code was generated at line 14. You cannot use the **go** command to go to a line with no code generated.

go Continues execution until a breakpoint is encountered or the program exits
(continued)

```
go sort
    continues program execution to the first line of the sort function.
```

```
go \myfile.c\sort 14
    continues program execution to line 14 of the sort function in
    module myfile.c. This example also assumes that code was
    generated at line 14.
```

```
go mylib.library:myfunc
    continues program execution to the myfunc function in the shared
    library mylib.library.
```

```
go $ after(5)
    continues program execution until the fifth time the current line is
    executed.
```

```
go 0x804A
    continues program execution until absolute address 0x804A is
    reached.
```

See Also **break**, **proceed**, **ps**, **trace**, **ts**

help Displays help information

Synopsis **h[elp]** [*command*]
 ? [*command*]

Description The **help** command displays information about commands and operands. If the *command* parameter is omitted, the Help window opens and a list of commands and topics is displayed. Entering one of the items on this list as the value for the *command* parameter displays information about that item.

 An alternate method of selecting an item is to double-click on the command or topic in the Help window. This method of selection can also be used from the help screens themselves. For example, at the bottom of the screen for the **dump** command there is a section that directs you to the **display** and **dzero** commands for additional information. You can access the help screen for either of these commands by double-clicking on the name of the command.

Examples **help**
 displays a list of commands and topics.

h break
 displays information about the **break** command.

? break
 displays information about the **break** command.

hunks Lists the addresses and sizes of all hunks

Synopsis **hu**[nks]

Description The **hunks** command displays the list of loaded hunks, their addresses, and their sizes for the current executable file.

The following example shows the output from the **hunks** command:

```
>hunks
Hunk  Address      Size
  0    00C32AE8    0xB54 (2900)
  1    00C28570    0x27C (636)
```

The size is displayed first in hexadecimal format followed by decimal format in parentheses.

Example **hunks**
lists all hunks for the current executable file.

jump Changes the current execution point

Synopsis `j[ump] location`

Description The **jump** command updates the program counter to point to a new location. The effect of using this command is that the code between the previous execution point and the new location is not executed. Any value for the *location* parameter that is valid for the **break** or **go** commands can be used. It is important to note that the stack is not modified in any way.

You should be very careful when using the **jump** command because certain registers may not be set up the way the code that is jumped to expects. Using the **jump** command is more likely to be safe if you always jump from one C source file line to another in the same function (not from assembly lines), and you have compiled with the **debug=symbolflush** or **debug=fullflush** options to make sure that registers are flushed to memory at C source line boundaries. If the **jump** command is used on optimized code, the registers are not likely to be set correctly.

Note: To avoid confusion, the **jump** command does not use the default radix setting. You must specify **0x** for an address; otherwise, you will be specifying a line number. For example, even if the radix is set to **hex**, the following command specifies a **jump** command for line 20 and not address **0x20**:

```
j 20
```

Examples **jump 12**
 jumps to line 12 in the current function.

```
jump 0x804E  
              jumps to the address 0x804E.
```

See Also **env**, **go**, **return**

list Lists the lines in a source file

Synopsis `list`
`list` `$`
`list` `+` `|` `-` *integer* [`L` *length*]
`list` *line-range*
`list` [*image:*]*module* [*line-range*]
`list` [[*image:*]*module*\]*function* [*line-range*]

Description The `list` command is used when debugging in line mode to list the lines in a source file. The current list line is set to the source line at which the program being debugged is stopped each time the user regains control.

The `list` command with no arguments displays lines starting with the current list line. The number of lines displayed is controlled by the `opt list` command.

The `$` parameter is used to list a number of lines above and below the current line at which the program is stopped. The number of lines above and below is controlled by the `opt context` command.

The *module* parameter specifies the name of a module in the program. A range of lines is specified with the *line-range* parameter, which can have either of the following forms:

line-1 [`[..]` *line-2*]
line-1 `L` *length*

In the first form the *line-range* parameter specifies a range of lines starting at *line-1* and ending at *line-2*. In the second form a range of lines is specified as starting at *line-1* and extending for *length* number of lines.

If the `list` command is given an environment (*module* or *function*), then the source file for that environment is opened, if possible, and the desired line range is displayed. If no line range is given, and no function is given, the listing begins at line 1. If a function name is given and no line range is given, lines are listed beginning with the function declaration.

Each time the `list` command is used, the current list line is set to the line immediately following the last listed line. Remember that the current list line is also updated each time the client program is allowed to execute.

Examples `list`
 lists the next group of lines. The number of lines listed is set by the `opt list` command.

list Lists the lines in a source file
(*continued*)

list \$
lists lines around the line pointed to by the current program counter.

list module
lists first lines of the specified module.

list \module 20 40
lists lines 20 through 40 of the specified module.

list \module 20 1 10
lists 10 lines of the specified module starting at line 20.

list +20
starts listing lines 20 lines down from the current line.

list -20
starts listing lines 20 lines up from the current line.

list -20 L 10
lists 10 lines, starting 20 lines up from the current line.

list foo
starts listing at the declaration of the function named **foo**.

list foo 10 20
lists lines 10 through 20 of the function named **foo**.

See Also **opt, source, unassemble**

log Logs debugger commands to a file

Synopsis `1o[g] [log-command]`

Description The `log` command is used to save to a file a record of all activity in the Dialog window including commands and results. Menu selections are not saved by the `log` command. This information is stored in a log file. The name of the file is set with the `log file` command. The default filename is `cpr.log`.

When issued without the *log-command* parameter, the `log` command can be used to display the current log state, either `on` or `off`, and the filename of the log file. The *log-command* parameter can be any of the following:

append

appends log information to the end of the log file

file filename

specifies the filename of the log file

off

closes the log file

on

opens a new log file

snap

snapshots contents of the Dialog window to the log file.

If a command that enables the logging of your session, such as `log on`, `log append`, or `log snap`, is issued when logging is already enabled, the results of the command are added to the current log file. If the log filename is changed while logging is enabled, the current log file is closed, and a new one is opened. The log file is automatically closed when you exit CodeProbe.

Examples

```
log snap
    saves the dialog for the current session into the log file named
    cpr.log.

log file foo.log
    sets the log file to the filename foo.log.

log on
    enables the log file.
```


log Logs debugger commands to a file
(*continued*)

log off
disables the log file.

log append
enables the log file and appends new log information to the end of
the log file.

opt Shows option values or changes the value of a debugger option

Synopsis **op**[t]
 op[t] **au**[toswap] [on | off]
 op[t] **ar**[rdim] *integer*
 op[t] **b**[adchar] *integer*
 op[t] **ca**[se] [on | off]
 op[t] **cat**[ch] [on | off]
 op[t] **co**[ntext] *integer*
 op[t] **dev**[ices] [on | off]
 op[t] **e**[cho] [on | off]
 op[t] **ib**[ytes] [on | off]
 op[t] **ig**[norepath] [on | off]
 op[t] **l**[ist] *integer*
 op[t] **rad**[ix] [d [ecimal] | h[ex]]
 op[t] **ran**[gelen] *integer*
 op[t] **re**[slib] [on | off]
 op[t] **se**[arch] [+ | -] *directory* [, *directory* [...]]
 op[t] **so**[urce] c | a[sm] | m[ixed] | n[ext]
 op[t] **st**[rlen] *integer*
 op[t] **t**[ask] [*task-ID*]
 op[t] **u**[nassemble] *integer*

Description The **opt** command is used to show the value of debugger options or to change them. The **opt** command by itself shows the current settings of all options. The following are the options and their possible values:

autoswap

if set to **on**, pushes the screen containing the applications output to the front each time control is given to the application. If set to **off**, the application screen is not pushed to the front. The default is **off**.

arrdim

controls the maximum number of array elements that are displayed at one time with the **display** command. The default is 20.

badchar

controls the number of non-ASCII characters that a **dzero** or **display** command will accept when displaying a string referenced by a character pointer. The default is 3.

case

if set to **on**, causes the **search** command to perform case-sensitive string searches. The default is **off**.

opt Shows option values or changes the value of a debugger option
(*continued*)

catch

if set to **on**, catches any new task generated by a task under debugger control. The default is **off**.

context

controls the number of lines displayed in the Source window preceding the current line. The default is 2.

devices

if set to **on**, attaches to, or catches, any new device process or task generated by a task under debugger control. The default is **off**.

echo

if set to **on**, causes the debugger to echo all commands to the Dialog window. This is particularly useful when using the **execute** command to execute a file of debugger commands. The default is **off**.

ibytes

if set to **on**, causes the debugger to display the instruction bytes being disassembled. This affects the output of the **unassemble** command. In windowing mode, it also affects the Source window in **mixed** and **asm** source modes. The default is **off**.

ignorepath

if set to **on**, causes the debugger to ignore the pathname for the source file provided by the compiler. The default is **off**.

list

controls the default number of lines displayed by the **list** command in line mode. The default is 6.

search

defines a set of directories to search for the source code. If no **+** or **-** is specified, the list of directories replaces the current list. Specifying a **+** at the beginning of the list appends the new list to the end of the current list. Specifying a **-** at the beginning of the list removes the specified directories from the current list.

source

controls the display in the Source window in windowing mode. Repeatedly choosing the **next** values cycles through the possible display types in the order **C**, **asm**, and **mixed**. The **source** option controls the output when you execute a **step** or **go**. The default is **C**.

opt Shows option values or changes the value of a debugger option
(continued)

strlen

controls the maximum number of bytes that are displayed when a character string is displayed with the **dzero** command. The default is 128.

task

changes the current task to the task identified by the *task-ID* parameter. The *task-ID* parameter can be either the name of a task or the address of a task control block, as displayed by the **tasks** command.

radix

sets the default input type for constants. It is used to avoid having to type the initial **0x** on hexadecimal constants. The default is decimal.

rangelen

controls the default size of ranges when no range size information is available. The default is 64.

reslib

if set to **on**, steps into resident libraries while tracing or running an application with watch breaks. The default is **off**.

unassemble

controls the default number of instructions that are disassembled by the **unassemble** command in line mode. It has no effect in C mode or windowing mode. The default is 4.

Examples **opt**
displays the settings of all options.

opt unassemble 10
sets unassemble count to 10.

opt search /src,/test
sets search directories.

opt search +test2
appends a new search directory.

See Also **display, dzero, list, unassemble**

proceed Single-steps over function calls

Synopsis **p[roceed]** [*integer*]

Description In **C** mode, the **proceed** command causes the debugger to step through the number of source statements specified by the *integer* parameter. In **Mixed** or **Asm** modes, it steps through *integer* number of machine instructions. If a function call is encountered, the function is executed as if it were a single statement. If the *integer* parameter is omitted, it defaults to 1.

A **proceed** command is performed when you press the Return key without entering any commands in the Dialog window. The F6 key can also be used to enter the **proceed** command.

Examples **proceed**

steps 1 source statement in **C** mode or 1 machine instruction if in **Mixed** or **Asm** modes.

proceed 5

steps 5 source statements in **C** mode or 5 machine instructions if in **Mixed** or **Asm** modes.

See Also **go**, **ps**, **trace**, **ts**

ps Single-steps over function calls by source line

Synopsis **ps** [*integer*]

Description The **ps** command causes the debugger to single-step through the number of C source statements specified by the *integer* parameter even if mode is set to **Mixed** or **Asm**. If a function call is encountered during stepping, the function is executed as if it were a single statement. If the *integer* parameter is omitted, it defaults to 1.

Example **ps 2**
steps through two source lines regardless of the debugger mode.

See Also **go, proceed, trace, ts**

quit Terminates the debugger

Synopsis **q[uit]**

Description The **quit** command terminates the debugging session and returns to the operating system prompt.

When running in cross-development mode, the **quit** command only terminates the cross debugger (CPRX) on the host machine, leaving the kernel (CPRK) waiting for a new debug session.

Normally, CodeProbe forces the application being debugged to call the **exit** function before terminating the application. The debugger then exits. If CodeProbe was started with the **-startup** option, the debugger exits without forcing the application to call **exit**.

Example **quit**
terminates the debugging session.

register Displays or modifies registers

Synopsis `r[egister] [register [[=] expression]]`

Description If no arguments are given, the **register** command displays the current contents of all the machine integer registers. If a register name is specified by the *register* parameter, the contents of that register are displayed. If an *expression* parameter is specified in addition to the *register* parameter, the value of the expression is stored in the register. The **display** and **set** commands can also be used to display and modify the contents of the registers.

Examples **register**
displays all registers and flags.

register d5 = 30
sets register D5 to 30.

r d3 = index
sets register D3 to the value of **index**.

register a0 &x
sets register A0 to the value pointed to by **x**.

r a1
displays the value stored in register A1.

See Also **display**, **fregister**, **rflag**, **set**

restart Restarts the program being debugged

Synopsis **res**[**tart**] [*argument-list*]

Description The **restart** command causes the program you are debugging to be reloaded and executed up to the entry point of **main**, just as when the debugger was first invoked. (If the **-startup** option was specified when you invoked the debugger, execution will stop before the startup code is executed and not at the first line of the **main** function.) Because the program is reloaded, all static data is reinitialized. All breakpoints will be retained and watch breaks on static and external variables will be retained and disabled.

The *argument-list* parameter provides the arguments for the program being debugged. These arguments are delimited by white space, just as would appear on the command line when invoking the program. The program will use these arguments as if it were invoked from the command line with them.

The **restart** command cannot be used with the **-command** command-line option or as one of a sequence of commands separated by semicolons. When used, the **restart** command must occur at the end of a command line.

The **restart** command potentially can cause your machine to crash if used improperly. When you enter the **restart** command, your application is immediately reloaded and re-executed with no cleanup of your currently running program other than calling the **exit** function to close files. This can leave the machine in an unacceptable state that can cause problems later. To avoid this problem, be sure to allow your application to run to completion if possible and do any necessary cleanup before issuing the **restart** command.

The **start** and **restart** commands are the same if the same arguments are provided by the *argument-list* parameter. The significant difference between the two commands is their behavior when they are specified with no arguments. The **start** command restarts the program without any arguments; however, the **restart** command with no arguments restarts the program with the original arguments used when the debugger was invoked.

restart Restarts the program being debugged
(continued)

Examples **restart**
restarts the program using the same arguments with which it was initially invoked.

restart myfile.txt 5
restarts the program passing it two arguments: `myfile.txt` and 5.

restart "sample string"
restarts the program passing in a string as an argument.

See Also **start**

return Returns immediately from the current function

Synopsis **ret**[urn] [*expression*]

Description The **return** command causes the current function to return to its caller without executing the rest of the function.

The value of the *expression* parameter is used as the return value from the function. The value is converted to the appropriate type if necessary; however, it must be a scalar quantity (not a structure or union).

An attempt to return a value from a function declared **void** is treated as an error, and a suitable message is displayed. Similarly, an error message is displayed if you use a simple return command (without a parameter) from within a function declared as having a return value. In all other cases, the return value is cast (if necessary) to the correct type and returned to the calling function.

When a **return** command is given, the current function is returned immediately to the calling function. The return value, if any, is returned to the calling function, and execution is suspended as though a breakpoint were triggered in the calling function after the call was made. For instance, note the following code fragment:

```
i = 5;
j = func(i);
if (j < 2)
    j++;
```

Assume that the **return** command is issued during the execution of **func** in the form **ret 7**. Execution stops inside the line that did the call, just before the assignment to **j**. You can then single-step to the next statement.

Examples **return stringptr**
returns the value of **stringptr**.

ret 5
returns an integer value of 5.

return
returns from a void function.

rflag Displays or modifies flags

Synopsis **rf[lag]** [*flag* ...]

Description The **rflag** command can be used to display or modify the settings of the machine flags. If no parameters are specified, the current settings of the flags are displayed. With one or more arguments, the indicated flags are set or cleared according to the values given. The flag names are specific to the target machine.

A flag setting can be one of the terms in the following table. The Set column comprises the terms used to set flags, while the Clear column comprises the terms used to clear flags.

Flag Name	Set	Clear
Overflow (yes/no)	OV	NV
Sign (negative/positive)	NG	PL
Zero (yes/no)	ZR	NZ
Auxiliary Carry (yes/no)	AC	NA
Carry (yes/no)	CY	NC

Examples **rflag**
displays all registers and flags.

rflag cy
sets the carry flag to 1.

See Also **fregister**, **register**

search Searches for a string in the current source file

Synopsis **sea**[rch] [*string*]

Description The **search** command searches the current source file for the specified *string*. In windowing mode, the search begins from the top line of the current Source window. In line mode, the search begins at the first line that would be displayed if a **list** command were entered with no arguments.

In windowing mode, the cursor is positioned at the beginning of the matching string. The Source window is also repositioned so that the line containing the match is the number of lines specified by the **context** option from the top of the window. The **context** is set by the **opt** command.

In line mode, the line containing the match is displayed.

The **search** command with no arguments causes the debugger to search for the string specified in the last search argument, starting from the new current line. If no matches are found, a message is printed. If the user then enters **search** again, the search is resumed beginning at the top of the file. The case-sensitivity option, **opt case**, determines whether the search is case sensitive or not.

Examples **search foo**
 finds the string "foo".

search foo;
 finds the string "foo". The semicolon used in this command is a command delimiter and is not considered part of the search string.

search foo\;
 finds the string "foo;". The escape character (\) causes the semicolon to be part of the search string.

search
 repeats the last search.

See Also **opt**, **list**

set Modifies the values of variables or memory locations

Synopsis **se[t]** (*variable* | *register*) = *expression*
 se[t] *address* [=] *string*

Description The **set** command is used to modify the values of variables, registers, or memory locations in the program being debugged.

You are most likely to use the first form of the **set** command in debugging C programs. It allows you to change the value of a variable in your program by referring to that variable by name. In this context, *variable* has a rather broad sense that includes references to array elements, structure members, and indirect references.

The *variable* parameter cannot refer to an aggregate such as a structure, union, or array.

The *register* parameter can be any valid register name, D0 through D7 and A0 through A7, and if a co-processor is present, FP0 through FP7. A **\$** prefix optionally may be used to distinguish the register from a variable of the same name.

The *expression* parameter is the same as used in the C assignment statement.

If necessary, the value for *variable* will be converted to the proper type in accordance with the normal C conversion rules; it is then entered into memory at the address of the first variable operand. This is similar to a C assignment statement for a scalar variable.

The second form shown in the synopsis is used to store a string into memory. The null terminator is not copied unless it is explicitly present in the string. The equal sign is optional. Support for this form of **set** command may be removed in future releases. The built-in **strcpy** and **memcpy** functions are preferred.

Examples **set i = 5**
 sets the variable **i** equal to the integer value 5.

set a = 3.1415
 sets the variable **a** equal to the floating-point value 3.1415.

set \$d0 = 300
 sets register D0 equal to 300. The **\$** is optional.

set time->date = newtime
 sets the **date** member of the **time** structure equal to the value of the variable **newtime**.

set stringptr "this string has no null byte"
 stores a string that is not null-terminated to the memory location pointed to by **stringptr**.

set Modifies the values of variables or memory locations
(*continued*)

```
set stringptr "this string has a null byte\0"
    stores a null-terminated string to the memory location pointed to by
    stringptr.
```

```
set stringptr "string one\0string two\0"
    stores two null-terminated strings to the memory location pointed to
    by stringptr.
```

See Also `fregister`, `memcpy`, `strcpy`, `register`

sleep Pauses for the time specified

Synopsis **sleep** *number*

Description The **sleep** command pauses for the number of seconds specified by the *number* parameter.

Example **sleep 10**
pauses for 10 seconds.

source Displays the source file

Synopsis **so[urce]** [*filename*]

Description The **source** command overrides the C source file associated with the current module. The specified value for *filename* replaces the filename associated with the module for the rest of the current debugging session. The new file is displayed in the Source window immediately, and its name is echoed in the Dialog window.

 If no *filename* parameter is specified, the name of the current source file is echoed in the Dialog window but the current source file is not changed.

 If the filename you want to change to is the same as the default filename but the path is different, you can use the **opt search** command instead of the **source** command to change the locations that CodeProbe looks for source files.

Examples **source**
 displays the current filename.

source test/test1.c
 changes the filename for the current module to **test1.c** in the **test** directory.

See Also **opt**

start Restarts the program being debugged

Synopsis **sta[rt]** [*argument-list*]

Description The **start** command causes the program you are debugging to be reloaded and executed up to the entry point of **main**, just as when the debugger was first invoked. (If the **-startup** option was specified when you invoked the debugger, execution will stop before the startup code is executed and not at the first line of the **main** function.) Because the program is reloaded, all static data are reinitialized. All breakpoints are retained and watch breaks on static and external variables are retained and disabled.

The *argument-list* parameter provides the arguments for the program being debugged. These arguments are delimited by white space, just as would appear on the command line when invoking the program. The program will use these arguments as if it were invoked from the command line with them.

The **start** command cannot be used with the **-command** command-line option or as one of a sequence of commands separated by semicolons. When used, the **start** command must occur at the end of a command line.

The **start** command potentially can cause your machine to crash if used improperly. When you enter the **start** command, your application is immediately reloaded and re-executed, with no cleanup of your currently running program other than calling the **exit** function to close files. This can leave the machine in an unacceptable state that can cause problems later. To avoid this problem, be sure to allow your application to run to completion and do any necessary cleanup before issuing the **start** command.

The **start** and **restart** commands are the same if the same arguments are provided by the *argument-list* parameter. The significant difference between the two commands is their behavior when they are specified with no arguments. The **start** command restarts the program without any arguments; however, the **restart** command with no arguments restarts the program with the original arguments used when the debugger was invoked.

start Restarts the program being debugged
(*continued*)

Examples **start**
 restarts the program being debugged without passing it any
 arguments.
 start myfile.txt 5
 restarts the program being debugged passing in the arguments
 myfile.text and 5.

See Also **restart**

symbol Finds the symbol nearest to the specified address

Synopsis **syb[ol]** *address*

Description The **symbol** command causes the debugger to search the symbol information and display the name of the symbol whose location is closest to the address specified by the *address* parameter.

Example **symbol 0x8040**
 displays the symbol whose address is nearest to **0x8040**.

See Also **symload**

symload Reads symbol information from an executable file

Synopsis **sym[load]** *executable-file*
 sym[load] **proc** *process-id*
 sym[load] **seg** *address executable-file*

Description The **symload** command associates an executable file and any debugging information found in that file with a loaded image.

 If given the filename of an *executable-file*, the **symload** command associates that file with the current hunk list.

 If the **proc** keyword is used, followed by the name of a process that is identified by the *process-id* parameter, the **symload** command attempts to map the segment list for the process with its executable file.

 If the **seg** keyword is specified along with an *address* and the filename of an executable file, the **symload** command treats the address as a pointer to a BCPL segment list and it maps the list to the executable file.

Examples **symload myapp**
 reads the current segment list and symbols from the file named **myapp**.
 sym proc myapp
 reads the segment list and executable file for process **myapp**.
 symload proc 0xC85428 myapp
 reads the segment list and executable file for the process located at **0xC85428**.
 symload seg 0xC84200 myapp
 uses the segment list at address **0xC84200**.

See Also **hunks**

tasks Displays system tasks

Synopsis **ta**[sks] [**a**[11]]

Description The **tasks** command displays information about tasks. By default, it displays the tasks currently under debugger control. If the **a11** option is specified, all system tasks are displayed.

The **opt task** command can be used to change to a new task.

Examples **tasks**
displays a listing of all tasks under debugger control.
ta a11
displays a listing of all tasks in the system.

See Also **activate, catch, deactivate, detach, opt**

trace Single-steps into function calls

Synopsis **t[race]** [*integer*]

Description In **C** mode, the **trace** command steps the number of source statements specified by the *integer* parameter. In **Mixed** or **Asm** modes, it steps the number of machine instructions specified by the *integer* parameter. If a function call is encountered, stepping continues into that function. If the *integer* parameter is omitted, **trace** will step one line.

Examples **trace**
 steps one source statement if in **C** mode, or one machine instruction if in **Mixed** or **Asm** mode.

t 5
 steps five source statements or machine instructions.

See Also **go**, **proceed**, **ps**, **ts**

ts Single-steps by source line into function calls

Synopsis **ts** [*integer*]

Description The **ts** command is similar to the **trace** command in that it will step into functions encountered during execution. It is different in that **ts** will only step by C source line, even if the source mode is set to **Asm** or **Mixed**. The *integer* parameter specifies the number of C source lines to step. If no parameter is specified, **ts** will step one line at a time.

Examples **ts**
 steps one source line regardless of the mode the debugger is in.
ts 2
 steps two source lines.

See Also **proceed, ps, trace**

unalias Deletes an alias

Synopsis **una[lias]** *name*
 una[lias] *****

Description The **unalias** command removes entries from the debugger's list of aliases. The *name* parameter refers to any alias name. You can use the ***** parameter as a wildcard character to remove all aliases.

Examples **unalias foo**
 deletes the alias for **foo**.

 una *
 deletes all aliases.

See Also **alias, define, undefine**

unassemble Displays memory as assembler instructions

Synopsis `u[nassemble] [location-1 [location-2]]`

Description The **unassemble** command displays the range of memory from *location-1* to *location-2* in assembler format.

By default, the command begins disassembling at the current program counter. If subsequent **unassemble** commands are performed before the application is given control by a **trace**, **proceed**, or **go** command, it will continue disassembling from the point at which it finished the last invocation.

If source line number information is available for the range of memory being disassembled, **unassemble** will display the C source line before displaying the corresponding instructions. It will display the number of assembler instructions generated for one C source line.

If no source information is available for the range of memory being displayed, **unassemble** will display the default number of instructions as specified by the **unassemble** default item in the Options menu. By default this number is 4 instructions.

The output of the **unassemble** command is equivalent in format to assembler mode output displayed in the Source Window. It will consist of three or four columns depending on the setting of the instruction bytes option, which is controlled by the **opt bytes** command. (See Chapter 4, “Setting Up the Debugging Environment,” for more information about setting instruction bytes.) The following is an example of **unassemble** output with instruction bytes on and source unavailable:

```
> unassemble
0x25F950 48E70130      MOVEM.L    D7/A2-A3, -(A7)
0x25F954 BFEC0004      CMPA.L    0004(A4), A7
0x25F958 65001BD6      BCS      00261530
```

The first field contains the hexadecimal address of the instruction being disassembled. The second field is a hexadecimal dump of the actual bytes composing the instruction. The third field consists of the M680X0 mnemonic for the given opcode. The fourth field consists of the operands to the instruction. If the instruction bytes option is turned off, the second field is not displayed.

Examples `unassemble \mod1\func1 10`
 unassembles line 10 of **func1** in **mod1**.
`unassemble 13 14`
 unassembles lines 13 to 14.

unassemble Displays memory as assembler instructions
(*continued*)

```
unassemble func1
    unassembles the first line of func1.
unassemble func1 func2
    unassembles from &func1 to &func2.
unassemble
    continues unassembling at the last location.
```

See Also **opt, list**

undefine Deletes a macro definition

Synopsis `[#]und[efine] name`
`[#]und[efine] *`

Description The **undefine** command removes entries from the debugger's list of macro definitions. The *name* parameter specifies the name of the macro to removed. You can use the `*` as a wildcard character to remove all macro definitions. The `#` sign can be inserted before the **undefine** command so that C header files (`.hfiles`) can be read by CodeProbe.

Examples `#undef foo`
deletes the macro named `foo`.

`undefine *`
deletes all macros.

See Also `alias`, `define`, `execute`, `unalias`

watch Sets a watch on a variable or memory

Synopsis `w[atch] expression | range [s[tatic] | d[ynamic]]`

Description The **watch** command is used to set a watch for the *expression* or memory *range* specified. Whenever control is returned to the user, for example by stepping or stopping at a breakpoint, the new value of the watched object is displayed in the Watch window. The Watch window can be opened with the **View** menu, the **window** command, or by pressing F1.

In line mode, the object is not automatically displayed. You must use the **wlist** command to display it.

The **watch** command does not cause the debugger to stop execution when the value being watched changes; it just displays the changed value. Use the **wbreak** command to cause the debugger to stop when a value is changed.

By default, watches are dynamic; that is, the expression being watched is re-evaluated whenever control returns to the debugger. The **static** option causes the watch expression to be evaluated once, when the watch is set. The result is treated as an address whose contents are displayed.

Examples

```

watch text->len
    sets a watch for the symbolic scalar text->len.

w \mod\f1\a[0] L 20
    sets a watch for a length of 20 bytes starting at the symbolic range
    specified by \mod\f1\a[0].

w &a[0] .. &a[5]
    sets a watch for the range from &a[0] to &a[5].

watch tmp[i] static
    if i = 3, watches tmp[3] regardless of changes to i.
  
```

See Also **wbreak, wclear, wdisable, wenable, wlist**

wbreak Sets a watch break

Synopsis **wb[reak]** *expression* | *range* [**s**[tatic] | **d**ynamic]]

Description The **wbreak** command sets a watch break for the *expression* or memory *range* specified. Whenever any byte in a specified range is modified, control is returned to you.

By default, watch breaks are static; that is, the expression is evaluated once when the watch break is set. The address result of the expression is then monitored by the watch break.

The **dynamic** option indicates that the watch break expression or range should be re-evaluated whenever control returns to the debugger.

Examples **wbreak text->len**

sets a watch break for the symbolic scalar **text->len**.

wb \mod\f1\a[0] L 20

sets a watch break for a length of 20 bytes starting at the symbolic range specified by **\mod\f1\a[0]**.

wbreak &a[0]..&a[5]

sets a watch break for the range from **&a[0]** to **&a[5]**.

wb p->q dynamic

sets a watch break for different locations depending on the value of **p**.

See Also **watch, wclear, wdisable, wenable, wlist**

wclear Clears one or more watches

Synopsis `wc[lear] integer [integer ...]`
`wc[lear] * | 1[last]`
`wc[lear] integer .. integer`

Description The **wclear** command is used to clear one or more watches or watch breaks. When a watch is cleared it ceases to exist and can be reinstated only by issuing the **watch** or **wbreak** command again. You can use the **wdisable** command to temporarily disable a watch.

The *integer* parameter specifies the number of a watch as displayed by the **wlist** command. Any number of *integer* values can be specified as parameters to the **wclear** command.

The *** parameter specifies that all watches and watch breaks should be cleared.

You can specify that all watches and watch breaks between two watch numbers be cleared by using the form *integer .. integer*.

The **last** parameter can be specified to clear the last watch or watch break that was set.

Examples `wclear 2 5 6`
clears watches or watch breaks numbered 2, 5, and 6.

`wclear last`
clears the last watch or watch break set.

`wc *`
clears all watches or watch breaks.

`wc4..7`
clears watches 4, 5, 6, and 7.

See Also **watch**, **wbreak**, **wclear**, **wdisable**, **wlist**

wdisable Disables one or more watches

Synopsis `wd[isable] integer [integer ...]
 wd[isable] * | l[ast]
 wd[isable] integer .. integer`

Description The **wdisable** command is used to disable the recognition of one or more watches or watch breaks. When a watch or watch break is disabled, it is no longer recognized, but it remains on the current list of watches. A **wenable** command can be used to re-enable a watch or watch break that has been disabled with the **wdisable** command.

The *integer* parameter specifies the number of a watch, as displayed by the **wlist** command. Any number of *integer* values can be specified as parameters to the **wdisable** command.

The *** parameter specifies that all watches and watch breaks should be disabled.

You can specify that all watches and watch breaks between two watch numbers be disabled by using the form *integer* .. *integer*.

The **last** parameter can be specified to disable the last watch or watch break that was set.

Examples `wdisable 2 5 6`
 disables watches or watch breaks numbered 2, 5, and 6.

`wdisable last`
 disables the last watch or watch break set.

`wdisable *`
 disables all watches or watch breaks.

`wdisable 4..7`
 disables watches 4, 5, 6, and 7.

See Also **watch**, **wbreak**, **wclear**, **wenable**, **wlist**

wenable Enables one or more watches

Synopsis **we[nable]** *integer* ...
 we[nable] [*****] | **1[ast]**
 we[nable] *integer* .. *integer*

Description The **wenable** command is used to enable the recognition of one or more watches or watch breaks that have been disabled by the **wdisable** command.

 The *integer* parameter specifies the number of a watch, as displayed by the **wlist** command. Any number of *integer* values can be specified as parameters to the **wenable** command.

 The ***** parameter specifies that all watches and watch breaks should be enabled.

 You can specify that all watches and watch breaks between two watch numbers be enabled by using the form *integer* .. *integer*.

 The **last** parameter can be specified to enable the last watch or watch break that was set.

Examples **we 2 5 6**
 enables watches or watch breaks numbered 2, 5, and 6.

we last
 enables the last watch or watch break set.

wenable *
 enables all watches or watch breaks.

we 4..7
 enables watches 4, 5, 6, and 7.

See Also **watch, wbreak, wclear, wdisable, wlist**

what is Determines the type of an object

Synopsis **wha**[**tis**] *expression*
 wha[**tis**] (*type*)

Description The **what is** command displays the type, location, and storage class of an object, or gives additional information about a data type.

If an *expression* is given, **what is** displays the C data type of the expression. If the result of the expression is a simple variable or a member of an array, **what is** displays the address of the object and identifies it as either **static**, **extern**, or **automatic**. If the result of the expression is a function, **what is** displays the address of the function, its return type, and its location.

If a parenthesized *type* is given, including types that are defined by a **typedef** statement, **what is** displays the full definition for the type. For **struct**, **union**, and **enum** types this means all members of the aggregate are displayed; for a type that has been defined by a **typedef** statement, this means the base type of the **typedef** is displayed. Using the **what is** command on basic C data types such as **int** or **long** is allowed, but it is not very useful.

Examples **what is i**
 determines the type of a variable named **i**.
wha main
 determines the type, address, and defining module for the **main** function.
wha 3.5
 determines the type of a constant.
what is Red
 determines the type of an enumeration constant.
wha *p->c[3]
 determines the type of the expression ***p->c[3]**.
what is (int) (1*3.4)
 determines the type of an expression with a type cast.
what is (node)
 determines the type of a **typedef**.
what is (struct X)
 displays the members of **struct X**.

where Shows the calling sequence

Synopsis **wh[re]** [**a[rgs]**] [*integer*]

Description The **where** command displays a backtrace of the function calls in the call sequence. Calls are listed in reverse order beginning with the most deeply nested function.

The **args** option causes the **where** command to display the arguments to each function in a manner similar to the **args** command.

The *integer* parameter specifies the number of calls to be displayed. By default, only the 20 most recent calls are printed.

Here is an example of the output from the **where** command:

```
> 1 In routine fpreg.o:\fpreg.c\five 48
> 2* Called from fpreg.o:\fpreg.c\six 68 (+0xE)
> 3 Called from fpreg.o:\fpreg.c\seven 82
> 4 Called from fpreg.o:\fpreg.c\main 92
> ...
```

The numbers on the left indicate a level number that can be passed to the **env** command. Level 1 is defined to be the function in which you are currently stopped or the run environment. The caller's level is 2, its caller is 3, and so on. These level number designations change over time as the program steps into and returns from functions. The asterisk indicates the current user environment as set by the **env** command.

The Calls window also displays this information.

Examples **wh**
displays a list of the last 20 function calls.

where 5
displays a list of the last 5 function calls.

where a
displays the arguments to each function.

See Also **args**, **env**

window Opens or closes a window

Synopsis **wi**[ndow] *window-name* [on | off]

Description The **window** command is used to open and close a specified window. Any of the following values can be specified as the *window-name* parameter:

c [alls]	mo [dules]	w [atch]
he [lp]	ms [g]	s [ource]
me [mory]	r [egister]	

If the **window** command is invoked with only a *window-name* parameter, the command acts as a toggle for the specified window; thus, if the window is currently opened, the **window** command will close it, and if it is currently closed it will be opened.

If **on** or **off** is specified, the window is forced into that state, regardless of its current state. If a window that is already open is set to **on**, it will pop to the top of the window hierarchy.

The **window** command is ignored in line mode.

Examples **wi register**
toggles the Register window.

window help on
opens the Help window or pops it to the top.

wlist Lists all watches

Synopsis **wl[ist]**

Description The **wlist** command lists all watches and watch breaks. Disabled watches are marked with an asterisk. Watch breaks are distinguished from regular watches by an exclamation mark. The value associated with the watch is displayed if available.

Example **wlist**
lists all watches.

See Also **watch, wbreak, wclear, wdisable, wenable**

wmsg Writes a message to the Message window

Synopsis **wmsg** *integer message-text*

Description The **wmsg** command is used to write the text specified by the *message-text* parameter in the Message window. The *integer* parameter specifies the location in the window for the text. Line 0 is the top line. This command is used primarily with debugger or AREXX scripts.

Examples **wmsg 0 --- This is the Message window. ---**
 displays the string **--- This is the Message window. ---**
 on the top line of the Message window.

wmsg 12 This goes on line 12.
 displays the string **This goes on line 12.** on line 12 of the
 Message window.

Built-in Functions

As described in Chapter 3, “Debugging Your Program”, built-in functions are standard, commonly used functions recognized by the compiler and expanded inline for efficiency. The SAS/C Compiler defines several built-in functions that are described in Chapter 4, “Using the System Header Files to Access Libraries,” of the *SAS/C Development System Library Reference*.

CodeProbe also defines several built-in functions. These functions are related to the library built-in functions in that several of them share the same names and the same functionality. The CodeProbe built-in functions are provided for the following reasons:

- for use with the `call` and `display` commands. Since the built-in functions provided with the compiler generate in-line code, these commands would not be able to access them.
- to add convenience and functionality. CodeProbe’s built-in functions provide functionality not available through other debugger commands.

The compiler’s built-in functions perform in accordance with the ANSI C standards. This may be slightly different from what CodeProbe’s built-in functions do. For example, passing `NULL` to a built-in function is an error in CodeProbe. Overlapping memory blocks may be handled differently. Also, the parameters to CodeProbe’s built-in functions are a little more flexible than the parameters to the C versions:

formal type	actual parameter types allowed
<code>void *</code>	constant, register, any pointer scalar, array, function, address, string constant
<code>char *</code>	constant, register, character pointer, unsigned character pointer, character array, string constant
<code>int</code>	constant, register, any integral scalar, bitfield, enumerated constant

Floating-point constants and registers are not allowed, and string constants can only be specified as the source operand.

Built-in Function Reference

This section describes the following built-in functions:

memcmp	compares two memory blocks
memcpy	copies a memory block (non-overlapping)
memmove	copies a memory block (possibly overlapping)
memset	sets memory to a specified value
strcat	concatenates two strings
strcmp	compares two strings
strcpy	copies a string
strlen	returns the length of a string.

memcmp Compares two memory blocks

Synopsis

```
i = memcmp(a, b, n);
int i;           /* comparison results */
void *a, *b;     /* blocks being compared */
int n;           /* block size in bytes */
```

Description The **memcmp** function compares two memory blocks and returns a value whose sign indicates the collating sequence of the blocks, as follows:

Return	Meaning
Negative	First block below the second
Zero	Blocks are equal
Positive	First block above the second

CodeProbe has a built-in **#define** that causes all references to **memcmp** to be referenced to **__builtin_memcmp**, which is the actual name of the CodeProbe function. If you want to step into or call an actual function called **memcmp** in your code, you should use the **undefine** command to clear the define or use the backtick character to escape the function's name.

Example

```
display memcmp(ptr1, ptr2, n)
    displays the result of comparing ptr1 and ptr2 for n bytes.

b 27 when(memcmp(ptr1, ptr2, 10) < 0)
    breaks at line 27 when the 10 bytes pointed to by ptr1 evaluate to
    a value less than the value of the 10 bytes pointed to by ptr2.
```

See Also **memcpy**, **memmove**, **memset**

memcpy Copies a memory block (non-overlapping)

Synopsis

```
to = memcpy(to, from, n);
void *to;    /* destination pointer */
void *from;  /* source pointer      */
int n;       /* block size in bytes */
```

Description The **memcpy** function copies data from one memory block to another. The destination pointer is returned.

The **memcpy** function cannot be used to copy overlapping memory blocks. You should use the **memmove** function when copying overlapping blocks. Also note that you can crash your machine if there is not enough memory to hold the data at the destination memory location.

CodeProbe has a built-in **#define** that causes all references to **memcpy** to be referenced to **__builtin_memcpy**, which is the actual name of the CodeProbe function. If you want to step into or call an actual function called **memcpy** in your code, you should use the **undefine** command to clear the define or use the backtick character to escape the function's name.

Example

```
call memcpy(ptr1, &j, 10)
    copies 10 bytes from ptr2 to ptr1.

call memcpy(0x804a, myptr, 50)
    copies 50 bytes from myptr to location 0x804a.

b 27 {memcpy(a0, a1, 15)}
    sets a breakpoint at line 27 with an action to copy 15 bytes from
    the memory pointed to by register A1 to the memory pointed to by
    register A0.
```

See Also **memcmp**, **memmove**, **memset**

memmove Copies a memory block (possibly overlapping)

Synopsis

```
to = memmove(to, from, n);
void *to;    /* destination pointer */
void *from;  /* source pointer      */
int n;       /* block size in bytes */
```

Description The **memmove** function copies data from one memory block to another. Overlapping blocks are handled correctly. The destination pointer is returned.

Note that you can crash your machine if there is not enough memory to hold the data at the destination memory location.

CodeProbe has a built-in **#define** that causes all references to **memmove** to be referenced to **__builtin_memmove**, which is the actual name of the CodeProbe function. If you want to step into or call an actual function called **memmove** in your code, you should use the **undefine** command to clear the define or use the backtick character to escape the function's name.

Example

```
call memmove(ptr1, &j, 10)
    copies 10 bytes from ptr2 to ptr1.

call memmove(0x804a, myptr, 50)
    copies 50 bytes from myptr to location 0x804a.

b 27 {memmove(a0, a1, 15)}
    sets a breakpoint at line 27 with an action to copy 15 bytes from
    the memory pointed to by register A1 to the memory pointed to by
    register A0.
```

See Also **memcpy**, **memcmp**, **memset**

memset Sets memory to a specified value

Synopsis

```
to = memset(to, c, n);
void *to    /* base of memory to be initialized */
int  c;     /* initialization value             */
int  n;     /* number of bytes to be initialized */
```

Description The **memset** function sets the specified number of bytes of memory to the specified value.

Note that you can crash your machine if there is not enough memory to hold the data at the destination memory location.

CodeProbe has a built-in **#define** that causes all references to **memset** to be referenced to **__builtin_memset**, which is the actual name of the CodeProbe function. If you want to step into or call an actual function called **memset** in your code, you should use the **undefine** command to clear the define or use the backtick character to escape the function's name.

Example

```
call memset(ptr, 0, 100)
    sets 100 bytes to 0 starting at the address pointed to by ptr.

call memset(a0, 'X', 50)
    sets 50 bytes to the ASCII character X starting at the address
    pointed to by register A0.
```

See Also **memcpy**, **memcmp**, **memmove**

strcat Concatenates two strings

Synopsis `to = strcat(to, from);`
 `char *to;        /* destination pointer */`
 `char *from;    /* source pointer        */`

Description The **strcat** function copies data from one string to the end of another string until a null character is found. Overlapping strings are not handled by the **strcat** function.

Note that you can crash your machine if there is not enough memory to hold the string at the destination memory location.

CodeProbe has a built-in **#define** that causes all references to **strcat** to be referenced to **__builtin_strcat**, which is the actual name of the CodeProbe function. If you want to step into or call an actual function called **strcat** in your code, you should use the **undefine** command to clear the define or use the backtick character to escape the function's name.

Example `call strcat(myptr, "foo")`
 concatenates the string "foo" after the string pointed to by the variable `myptr`.

See Also `strlen, strcmp, strcpy`

strcmp Compares two strings

Synopsis `i = strcmp(a, b);`
 `int i;            /* comparison result        */`
 `char *a, *b; /* strings being compared */`

Description The **strcmp** function compares two strings and returns a value whose signs indicate the collating sequence of the blocks as follows:

Return	Meaning
Negative	First string below the second
Zero	Strings are equal
Positive	First string above the second

CodeProbe has a built-in **#define** that causes all references to **strcmp** to be referenced to **__builtin_strcmp**, which is the actual name of the CodeProbe function. If you want to step into or call an actual function called **strcmp** in your code, you should use the **undefine** command to clear the define or use the backtick character to escape the function's name.

Example `display strcmp("abc", "def")`
 tests the strings "abc" and "def" and displays the return value.
 The value indicates whether the strings are equal or which string is higher.

`break myfunc when(strcmp(arg, "foobar") == 0)`
 breaks at the function **myfunc** when the variable **arg** points to the string "foobar".

See Also **strlen, strcpy, strcat**

strcpy Copies a string

Synopsis

```
to = strcpy(to, from);
char *to;      /* destination pointer */
char *from;    /* source pointer      */
```

Description The **strcpy** function copies data from one string to another until a null character is found. Overlapping strings are not handled by the **strcpy** function.

Note that you can crash your machine if there is not enough memory to hold the string at the destination memory location.

CodeProbe has a built-in **#define** that causes all references to **strcpy** to be referenced to **__builtin_strcpy**, which is the actual name of the CodeProbe function. If you want to step into or call an actual function called **strcpy** in your code, you should use the **undefine** command to clear the define or use the backtick character to escape the function's name.

Example **call strcpy(a, b)**
copies the string pointed to by **b** to the memory location pointed to by **a**.

See Also **strlen, strcmp, strcat**

strlen Returns the length of a string

Synopsis

```
len = strlen(s);
int len;    /* length of string s      */
char *s;    /* string to scan for length */
```

Description The **strlen** function returns the length of a string, as determined by the index of the first null character found.

CodeProbe has a built-in **#define** that causes all references to **strlen** to be referenced to **__builtin_strlen**, which is the actual name of the CodeProbe function. If you want to step into or call an actual function called **strlen** in your code, you should use the **undefine** command to clear the define or use the backtick character to escape the function's name.

Example

```
display strlen("hello")
    displays the length of the string "hello", which is 5.

display strlen(0x804a)
    displays the length of the string starting at address 0x804a.
```

See Also **strcmp, strcpy, strcat**



Part 2

Using the SAS/C[®]

Utilities

Chapter 10 Utility Reference

10 Utility Reference

This chapter describes the utilities included with the SAS/C Development System. The utilities are described in alphabetical order. The description of each utility includes each of the following sections, if applicable:

<i>Synopsis</i>	shows the format of the command that invokes the utility
<i>Description</i>	describes what the utility does and contains all the information you need to use the utility
<i>Example</i>	contains examples that illustrate how to use the utility
<i>Error Messages</i>	describes the error messages, if any, produced by the utility and the action required to fix the error.

cover Analyzes coverage data

Synopsis **cover** [*options*] [*datafile*]. . .

Description The **cover** utility analyzes the coverage data produced when you compile your program with the **coverage** option, link with the object file **covutil.o**, and run your program. Using this data, you can determine which lines of code in your program were executed and which were not. This information can help you design test cases that exercise all paths through your program.

When you run your program, it writes coverage data to the file **cover.dat** in the current directory. If this file does not exist, your program creates it. If this file already exists, your program reads in the existing **cover.dat**, merges in the new coverage data, and writes the merged data back to **cover.dat**.

This utility analyzes the coverage data file, locates your C source files, and produces new output files containing a copy of your C source with arrows pointing to lines that generated code that was not executed. The output file has the same basename as the C source file, but with the extension **.cov** instead of **.c**.

By default, **cover** looks for the coverage data file **cover.dat** in the current directory. If you choose to rename **cover.dat**, you must specify *datafile* when you run **cover**.

Your program must adhere to the following rules:

- The main routine must be called **main** rather than **_main** because **covutil.o** replaces **_main** with an enhanced version.
- Your program must end by calling **exit** or by falling through the end of **main**. (Do not call **XCEXIT** or **abort**, for example.)
- Your program must not be a resident library or device.

To use **cover**, follow these steps:

1. Compile your program with the **coverage** and **debug=line** options. (You can use a higher debug level if you like.) Do not specify the **nostdio** option. Specifying **coverage** slows down your program considerably, so do not specify **coverage** for production software. You can choose to compile only certain modules with **coverage** to reduce overhead.
2. Link your compiled program with **covutil.o**. This file is located in **sc:examples/cover**. The **covutil.o** file replaces the **_main** and **_exit** functions in your program with enhanced versions that can construct and write out the coverage data.
3. Run your program normally to create **cover.dat**.
4. Run the **cover** utility.

cover Analyzes coverage data
(continued)

The **cover** utility supports the following options:

dir=directory-pathname

specifies the directory path or paths to search to locate the C source files. The coverage data contains the name of the C source file, but not its directory. If the C source file is not in the current directory, specify the **dir=** option with the proper directory as its argument. To specify multiple pathnames, separate each pathname with a plus (+) sign or a comma (.). Also, you can specify **dir=** as many times as necessary.

merge filename

tells **cover** to read the specified data files, merge the information contained in them, and write the merged data to *filename*. If you specify **merge**, **cover** does not look for C source files and does not produce a report. This option allows you to combine the results of a suite of tests to determine what code is exercised by the test suite as a whole.

nosource

tells **cover** not to read in the C source files. Instead, it prints a report to the Shell window listing the names of all source files and the lines in each file that were not executed.

Example To analyze the coverage of the **lines.c** program, open a Shell and use the **cd** command to change to the **sc:examples/lines** drawer. Enter the following command:

```
sc lines.c sc:examples/cover/covutil.o coverage debug=line link
```

The compiler produces a program called **lines**. Run this program with no command-line options:

```
lines
```

Watch the lines for a few seconds, then click on the **close** gadget to terminate the program. The program produces a file called **cover.dat**. Run the **cover** utility on this file:

```
cover
```

cover Analyzes coverage data
(*continued*)

The **cover** utility reads **cover.dat** and **lines.c** and writes the file **lines.cov**. Use your editor to examine **lines.cov**. Lines that were not executed are highlighted with an arrow (**=>**) in the left margin.

diff Determines the differences between two files

Synopsis `diff [>destination] [options] file1 file2`

Description The **diff** utility determines the differences in contents between two files and describes the lines that you should delete from, add to, or change in *file2* to make it look like *file1*. To help you find those lines, **diff** identifies the line numbers and gives the text of the lines.

If you enter the **diff** command without specifying filenames, **diff** displays the version that you are using and describes the options available.

Output from this utility is sent to the standard output device, unless you redirect it by specifying the **-o** option or by entering a greater than (>) sign followed by a destination.

diff supports the following options:

- bnn** sets the size of the I/O buffer to the value specified by *nn*. The default I/O buffer size is 4K, and the maximum size is limited by the amount of available system memory. Increasing the size of the I/O buffer makes **diff** run faster.
- c** displays only those lines common to both files.
- Fnn** sets the column number where you want **diff** to begin comparing lines. For example, if you set *nn* to 25, **diff** ignores the first 25 characters on each line.
- Lnn** sets the column number where you want **diff** to stop comparing lines. For example, if you set *nn* to 75, **diff** ignores the characters beyond column 75 on each line.
- lnn** defines the number of lines, *nn*, per file that can be handled by **diff**. **diff** uses two tables (one for each file) to keep track of the lines read in from each file. The default size of each of these tables is 2000; the maximum number is limited by memory availability. If **diff** is running out of memory and your files are less than 2000 lines in length, you can use this option to increase the amount of memory available, or you can check to see if any other tasks are running concurrently and shut down any unwanted tasks.
- ofile** sends the output to the specified file (or device). You can use this option instead of the AmigaDOS redirection command. You can send the output to a different device by specifying the device name. For example, to send output to the printer, specify **-opr t:.** To send output to a file on drive **df0:**, specify **-odf0:filename.**

diff Determines the differences between two files
(continued)

- p filters out unprintable characters from the input stream. You can use this option to clean out an AmigaDOS binary file.
- q suppresses any messages if there are no differences between the specified files.
- w ignores differences related solely to space or tab characters and reduces sequences of these characters to a single space. This option also removes trailing blanks. When you specify the -w option, **diff** uses additional memory to create a third table for each line in its compressed form.

Note: The appendix, “**diff** File-Matching Algorithm,” discusses the theory of file matching and how it affects **diff**.

To compare the two files, **newfile** and **oldfile**, you enter the following command:

```
diff newfile oldfile
```

diff compares the two files and gives you a set of instructions to follow to change **oldfile** into **newfile**. The instructions begin with the following message:

```
TO TRANSFORM oldfile INTO newfile ...
```

The instructions are displayed in three types of *blocks*: a delete block, an append block, and a change block.

A *delete block* describes the lines that you should delete from **oldfile**, and it has the following form:

```
*** DELETE [i,j] FROM oldfile ***
<Text of line 1 in oldfile.
<Text of line 2 in oldfile.
<Text of line 3 in oldfile.
```

The numbers *i* and *j* are the first and last line numbers in **oldfile** of the lines to be deleted. A less than (<) sign preceding a line of text indicates that you should delete the line.

diff Determines the differences between two files
(continued)

An *append block* describes the lines that you should add to **oldfile**, and it has the following form:

```
*** APPEND AFTER i IN oldfile ***
>Text of first line that you should add.
>Text of second line that you should add.
>Text of third line that you should add.
```

The number *i* is the line number in **oldfile** after which the lines should be added. A greater than (>) sign preceding a line of text indicates that you should add the line.

Note: The line number *i* will not be correct if you make other changes to the lines in **oldfile** before line number *i*.

A *change block* lists the lines in **oldfile** that you should change and shows how to change them. **diff** presents this information as a series of lines to delete and a series of lines to add in place of the deleted lines. A change block has the following form:

```
*** CHANGE [i,j] IN oldfile TO [m,n] IN newfile ***
<Line 1 in oldfile that you should change.
<Line 2 in oldfile that you should change.
<Line 3 in oldfile that you should change.
-----
>New text (from newfile) for line 1 in oldfile.
>New text (from newfile) for line 2 in oldfile.
>New text (from newfile) for line 3 in oldfile.
```

The numbers *i* and *j* are the first and last line numbers in **oldfile** of the lines to be deleted. The numbers *m* and *n* are the first and last line numbers in **newfile** of the lines you need to add to **oldfile**.

Examples Suppose you wanted to compare the contents of the files **cow** and **lamb**. The file **lamb** contains the following lines:

```
Mary had a little lamb.
Its fleece was white as snow.
Everywhere that Mary went,
The lamb was sure to go.
```

diff Determines the differences between two files
(continued)

The file `cow` contains the following lines:

```
Mary had a very large cow.
Its fleece was white as snow,
Mary had a what?
Everywhere that Mary went,
```

If you entered the command

```
diff lamb cow
```

this utility would produce the following:

```
TO TRANSFORM cow INTO lamb ...

*** CHANGE 1 IN cow TO 1 IN lamb ***
< Mary had a very large cow.
-----
> Mary had a little lamb.

*** DELETE 3 FROM cow ***
< Mary had a what?

*** APPEND AFTER 4 IN cow ***
> The lamb was sure to go.
```

Error Messages The `diff` utility may generate the following error messages:

Can't open file

indicates that `diff` cannot open the named file. The file may not exist or it may be protected.

Diff I/O Error

indicates an I/O problem.

Improper -l specification: nn

indicates that `diff` cannot interpret the `nn` value specified with the `-l` option as a number.

diff Determines the differences between two files
(continued)

Internal Error: ...

indicates an internal error. Please contact the Technical Support Division at SAS Institute if you see this error message. The Technical Support representative will need the following information:

- ☐ the version of **diff** you are using
- ☐ the exact wording of the command line you used
- ☐ the exact wording of the message you received
- ☐ the size of the files you were trying to compare.

Line table overflow

indicates that one of the two line tables, in which **diff** stores the lines from each file while they are being compared, is too small to hold the lines in one of the files. Try increasing the line table size with the **-l** option.

Not enough memory for nn lines per file

indicates that you have used the **-l** option to increase the line table size, but you do not have enough memory to increase it by the amount you specified.

Out of memory

indicates that **diff** is out of memory. Try decreasing the line table size with the **-l** option if possible. Do not use the **-w** option. You can also try decreasing the size of the I/O buffer by using the **-b** option; however, this action makes **diff** run slower because more disk accesses may be required. If you are using a hard disk, this factor may be negligible.

Too many file names: ...

indicates that you have made an error on the command line in such a way that **diff** assumes you are attempting to operate on more than two files.

Unrecognized option

indicates that you specified an option that **diff** does not recognize.

grep Searches for and prints regular expressions

Synopsis **grep** [>destination] [options] pattern file . . .

Description The **grep** utility searches the files you specify for all the lines that contain the specified pattern. For each file in which it finds the pattern, **grep** displays, on the screen, the filename, line number, and contents of the line that contains the matching string. The patterns that you can specify can be specific or general, and they are sometimes referred to as *regular expressions*. In this book, they are referred to as *patterns*.

For example, suppose you are working on a long document in a file named **document.txt**. You originally were going to call the document a user manual, but you have decided to call it a user's guide. You need to see where you have used the word *manual* to describe the document. You could enter the following command:

```
grep manual document.txt
```

grep displays all of the lines in the file that contain the word *manual*. These lines could include such phrases as *the manual*, *insert the paper manually*, or *refer to your AmigaDOS manual*.

By using special characters as described under “Specifying the Pattern” later in this discussion, you can tell **grep** exactly what to look for so that it does not display lines in which you are not interested.

To specify the files that you want **grep** to search, you can use the AmigaDOS wildcards (#?). For example, to search all files in the current directory that have an extension of **.c**, enter **#?.c** as the filename. To search all files in the root level directory of drive **df0:** that begin with the letter **g** and have the extension **.c**, enter **df0:g#?.c**. AmigaDOS ignores the case of filenames. It will find the files you specify whether you enter the name in uppercase, lowercase, or mixed case.

You can redirect the output from **grep** by specifying a greater than (>) sign followed by the destination where you want the output sent. You may want to redirect the output to a file and print that file.

grep Searches for and prints regular expressions
(continued)

grep supports the following options:

- c prints the total number of matched lines.
- f prints only the names of all files in which **grep** finds a string that matches the pattern.
- n tells **grep** not to display line numbers.
- p displays only printable ASCII characters. Non-printable characters are filtered out. This option is useful if you are searching a binary file that may contain control characters.
- q does not display filenames or line numbers.
- s displays the names of all files that **grep** searches. Normally, **grep** displays only those filenames in which it found a match for the pattern.
- v prints only the lines in which a match of the pattern is not found.
- V prints the version number of **grep**.
- \$ tells **grep** not to distinguish between upper- and lowercase. For example, if you use this option, the pattern **int** would match **int**, **INT**, **Int**, and **INT**.

Specifying the Pattern By using characters that **grep** interprets in special ways, you can specify some very complex patterns. Because **grep** is so flexible, you must also be specific about the pattern for which you are searching. The following list describes the special characters that you can use to specify the pattern for which you are looking:

- . is the **grep** wildcard character. A period will match any single character, including a space, tab, newline, or control character. A period followed by an asterisk(*) tells **grep** that any number of any characters can be present. The **.*** sequence is equivalent to the **#?** sequence for AmigaDOS commands.
- " " are used to enclose patterns that include space characters.

grep Searches for and prints regular expressions
(continued)

- [] are used to search for any one character from a set of characters by enclosing that set of characters inside square brackets. A set of characters enclosed in square brackets is called a *character class*. A character class will match one single character. For example, to search for lines that contain vowels, you can enter [aeiou].
 - is used to specify a range. Inside a character class, you can specify a range of characters by entering the first character in the range and the last character in the range separated by a hyphen. The first character must occur alphabetically before the second. For example, to find any one of the lowercase alphabetic characters, you enter [a-z].
 - ! is used as the negation character when included as the first character in a character class. An exclamation point tells **grep** to find any character that is not in the character class. For example, to find any one character that is not a lowercase alphabetic character, enter [!a-z]. To find any one character that is not a lowercase alphabetic character and is not the end-of-line character, enter [!a-z\n].
- * tells **grep** that the preceding character does not have to be present in the pattern but can be present an unlimited number of times. In other words, an asterisk means “zero or more of.”
A period followed by an asterisk tells **grep** that any number of any characters can be present. The .* sequence is equivalent to the #? sequence for AmigaDOS commands.
- + tells **grep** that the preceding character must be present in the pattern at least one time but can be present an unlimited number of times. In other words, a plus sign means “one or more of.” For example, to find all lines in your file that contain one or more of the lowercase letter t, enter t+.
- ^ tells **grep** that the pattern must begin in column 0. Enter the caret (^) at the beginning of the pattern. **grep** will look only for those lines in which the pattern occurs beginning in column 0. For example, to find all lines that begin with a left parenthesis, enter ^(.).

grep Searches for and prints regular expressions
(continued)

- \$ tells **grep** that the pattern must occur at the end of a line. Enter the dollar sign at the end of the pattern. For example, to find all lines that end with a right parenthesis, enter `)$`.
- \ is the escape character. If you want to search for any of the special characters included in this list, you must enter a backslash before that character. For example, to find all lines that contain a caret, enter `\^`. Similarly, if you want to search for a backslash, enter two backslashes, `\\`.

You also need to use the backslash to identify certain additional characters. These characters are as follows:

- `\b` backspace
- `\n` newline
- `\r` carriage return
- `\s` space
- `\t` tab
- `\xij` control character identified by hexadecimal digits *ij*

For example, to locate all lines containing the Control-g character, enter `\x07`.

You can combine these characters to specify any pattern you need. The way in which you specify the pattern to **grep** determines the response you will receive. **grep** does not convert tabs to spaces or change any combinations of spaces and tabs. If **grep** does not respond as you expect it to, try escaping any special characters with the backslash (`\`) character. Alternatively, try reducing the number of the special characters in your search pattern.

Examples To search for one word, you need only specify the word and the file in which you want to search. For example, the following command searches the file `document.txt` for the word *manual*:

```
grep manual document.txt
```

To search for a string of words that are separated by spaces you must enclose the entire string within double quotes, as follows:

```
grep "this is the string to search for" document.txt
```


grep Searches for and prints regular expressions
(continued)

Specifying Any Single Character

You can tell **grep** that a character can be any character by using a period (.) to designate that character. For example, you can specify a four-character pattern, beginning with **th** and ending with **n**, in which the third character can be anything, by specifying the following command:

```
grep th.n document.txt
```

This command will display all lines with patterns such as

```
than    then    thanks  Ethan
```

The period will match any single character, including an alphabetic character, numeric character, space, tab, newline, or control character. For example, you can enter the following command:

```
grep 123.45 document.txt
```

This command will display all lines containing any of the following strings:

```
123a45    123 45    123!45    123(45    123.45
```

Specifying Character Classes

A character class is useful when you need to look for characters that do not follow a pattern or when you do not care about the case of a character. For example, the following command displays all lines containing the symbols **%**, **&**, or **~**:

```
grep [%&~] document.txt
```

If you want to display all lines containing either **the** or **The**, you can enter the following command:

```
grep [Tt]he document.txt
```

grep Searches for and prints regular expressions
(continued)

Similarly, you can enter the following command:

```
grep [Tt][Hh][Ee] document.txt
```

This command displays all lines containing any of the following strings:

```
THE  The  The  the  thE  tHE  tHe  The
```

Using the `-$` option produces the same results:

```
grep -$ the document.txt
```

Inside a character class, you can use a minus (–) sign to specify a range of characters. The first character must occur alphabetically before the second. For example, the following command displays all lines that contain a lowercase letter:

```
grep [a-z] document.txt
```

Similarly, the following command displays all lines that contain any lowercase letter, any uppercase letter, or any decimal number:

```
grep [a-zA-Z0-9] document.txt
```

A practical use of character classes arises when you need to display all the lines in your file in which you refer to a specific year. Since each character class matches one character, enter the following command:

```
grep [0-9][0-9][0-9][0-9] document.txt
```

If you are concerned only with years in the nineteenth and twentieth centuries, then enter this command:

```
grep 1[89][0-9][0-9] document.txt
```

grep Searches for and prints regular expressions
(continued)

Also, the exclamation point (!) means *not* if it occurs as the first character in a character class. For example, the following command displays all lines that contain any character that is not a lowercase letter:

```
grep [!a-z] document.txt
```

The following command displays all lines that contain any character that is not the number sign (#):

```
grep [!#] document.txt
```

Specifying the Number of Times a Character Can Appear

An asterisk (*) tells **grep** that the preceding character can be present zero or more times. For example, you may have a file in which you have referred to *Gail* and mentioned a year in which some event has occurred. To display those lines in which both *Gail* and a year occur, enter the following command:

```
grep "Gail.*[0-9][0-9][0-9][0-9]" document.txt
```

The period (.) followed by an asterisk (*) tells **grep** that any character can be between *Gail* and the year and that you do not care how many of the characters appear. This command would display lines such as the following:

```
Call Gail Barbara Industries at 1-999-629-9310.
Gaillard de Marentonneau was a 1900s century botanist.
My eldest daughter Gail was born in 1969.
```

You can also use the asterisk (*) after a character class if you want to tell **grep** that you do not care how many of the characters in the character class occur in the pattern. For example, the following command displays any combination of zero or more lower- or uppercase letters **t** and the letters **he**:

```
grep [tT]*he document.txt
```

grep Searches for and prints regular expressions
(continued)

This command displays lines containing strings such as the following:

```
he tthe These hero penuche inherit mother
```

Using the plus (+) sign instead of the asterisk (*) tells **grep** that you want the string to include at least one **t**. Using a plus sign displays the following strings:

```
tthe These mother
```

If you want to display only lines that contain the word **the**, you can use the following command:

```
grep " [tT]+he " document.txt
```

Of the previously listed words, this command displays only **tthe**. If you enter an asterisk (*) instead of a plus (+) sign, this command displays **he** and **tthe**. If you include an exclamation point (!) as the first character in the character class, this command displays only the following strings:

```
he hero penuche inherit
```

Specifying a Pattern at the Beginning or End of a Line

As the first character in the pattern, a caret (^) tells **grep** to display only those lines that begin with the pattern. For example, to display only lines that begin with numerals, you can enter the following command:

```
grep ^[0-9] document.txt
```

Similarly, you can tell **grep** to display only those lines that end with a pattern by entering a dollar sign (\$) as the last character in the pattern. For example, the following command tells **grep** to display all lines that end with the word **the**:

```
grep " [tT]+he$" document.txt
```

grep Searches for and prints regular expressions
(continued)

Specifying Patterns that Contain Special Characters

Using a backslash (\), you can search for any of the special characters that **grep** recognizes. For example, if you want to search for a dollar sign (\$), enter a backslash (\) in front of the dollar sign. **grep** assumes you are looking for a dollar sign and not for a pattern that occurs at the end of a line. Similarly, the following command displays all lines that contain left brackets ([):

```
grep \[ document.txt
```

You also can use the backslash (\) to tell **grep** that you want to display all lines that contain backslashes, as follows:

```
grep \\ document.txt
```

To locate all lines containing tab characters, enter the following command:

```
grep \t document.txt
```

You also can use the backslash (\) inside of a character class. For example, to find all lines that begin with a space or a tab character, you can enter the following command:

```
grep "^[ \t]" document.txt
```

Because the period (.) is the **grep** wildcard character, to search for the phrase *and so on* . . ., use the following command:

```
grep "and so on \.\.\." document.txt
```

You can use `\xij` to search for control characters. Control characters are generally used as printer format instructions in files produced by word processors. Identify the character by its hexadecimal number (*ij*). For example, to display all lines that contain Control-g, enter the following command:

```
grep \x07 document.txt
```

grep Searches for and prints regular expressions
(continued)

To find lines that contain Control-z, enter the following command:

```
grep \x1a document.txt
```

The dollar sign (\$) special character works much like the carat (^) except that it forces the match to occur only at the end of a line. For example, to search for all occurrences of the C language comment terminator, */ , at the end of a line, enter the following command:

```
grep */$ document.txt
```

The backslash (\) tells **grep** not to interpret the asterisk (*) in its special sense as a closure operator.

Suppose you have a number of C language source files, each with filenames that end with .c, and you need to search these files to find all calls of either of the **read** or **fread** functions. Enter the following command:

```
grep "f*read *(" #?.c
```

This command prints every line in these files that contains either **read** or **fread**, followed by 0 or more spaces and a left parenthesis.

You also can search your files for all function definitions. Assume that each function definition begins with the function name and parameter list on a separate line. You can use the following command:

```
grep "^[a-zA-Z_ ]+[a-zA-Z0-9_ ]*(" #?.c
```

The pattern matches only expressions that begin a line. The expression must consist of one or more lowercase letters, uppercase letters, or the underscore (_), followed by 0 or more spaces, and then a left parenthesis. This command will print lines such as the following:

```
getl(s)          /* get a line */
_read ()
my_read(fp) { fread(fp,x);}
```

However, this command will not select a line such as

```
char *getl(p)    /* get a line */
```

grep Searches for and prints regular expressions
(continued)

You can use **grep** to help locate mismatching left and right braces ({}). To print each line that contains a left or right brace, you can enter the following:

```
grep [[]] #?.c
```

Error Messages **grep** may generate the following error messages:

Bad character class

indicates that **grep** is unable to interpret a character class in the pattern you have asked it to find. Check to see if you have used any special symbols that you did not intend to use. This message also can be generated if, in specifying a character class and using the minus (–) character to indicate a range of characters, the first character in the range does not alphabetically precede the second (for example, [d–a]).

Bad pattern

indicates that **grep** is unable to interpret the pattern you are asking it to find. You may have used a special character in the pattern that does not make sense in this context. Check to see if you have forgotten to escape a special character.

Can't find file(s) ...

indicates that **grep** is unable to find one or more of the files you have asked it to search. The files may not exist, or you may have misspelled the filenames.

Can't open ...

indicates that **grep** is unable to open the named file. The file may not exist or may be protected.

Closing] not found

indicates that **grep** believes that you have asked it to search for a pattern that contains a character class, but you have omitted the right bracket (]) that terminates the character class. If not, you may have included a left bracket ([) in the pattern but forgot to use a backslash (\) before it.

Empty character class

indicates that a character class you have used in your pattern has no members, for example [].

grep Searches for and prints regular expressions
(continued)

Improper hex specification

indicates that you have used the **x** escape sequence but have not followed it with two recognizable hexadecimal digits.

Incompatible combination of options

indicates that the options with which you have invoked **grep** are contradictory.

Internal Error: . . .

indicates an internal error. Please contact the Technical Support Division at SAS Institute if you see this message. The Technical Support representative will need the following information:

- ☐ the version number of **grep** you are using
- ☐ the exact wording of the command line you used
- ☐ the exact wording of the resulting error message.

Invalid option

indicates that you have used a command line option that **grep** does not recognize. This message also can be generated if your pattern begins with a minus sign (–) but you have not escaped it or enclosed the pattern in double quote marks (“).

No beginning double quote in pattern

indicates that **grep** has detected double quote marks (") in a pattern that did not start with a double quote. You may need a backslash (\) before the double quote.

No file arguments provided

indicates that **grep** believes you have specified a pattern but no filenames. This message also may be generated if you invoke **grep** with a filename but no pattern.

No pattern or file arguments given

indicates that **grep** cannot find either a pattern or filename on its command line.

Out of space

indicates that **grep** has run out of space in constructing its pattern representation. Please contact the Technical Support Division at SAS Institute if you see this message.

Pattern ill-formed: no terminating double quote

indicates that you started the pattern with a double quote (") but failed to terminate it with one.

grep Searches for and prints regular expressions
(*continued*)

Too few arguments to GREP

indicates that **grep** demands at least two arguments on its command line: a pattern and at least one filename.

gst Manages global symbol tables

Synopsis **gst** [*gst-filename*] [*options*] [*symbol-name* . . .]

Description A *Global Symbol Table* (GST) is a collection of all the information defined in a set of header files and stored in a format that the compiler can use easily and quickly. You can use GSTs to reduce significantly the processing time required for header files. However, GSTs require additional disk space and impose some restrictions on your header files. For information on creating and using GSTs, refer to Chapter 4, “Using the System Header Files to Access Libraries,” in *SAS/C Development System Library Reference*.

The **gst** utility helps you manage GSTs and extract information from a GST. With the **gst** utility, you can

- ☐ load a GST into memory
- ☐ unload a GST
- ☐ list all GSTs currently in memory
- ☐ extract symbol information from a GST.

GSTs must be loaded into memory before they can be used. Normally, a GST is loaded automatically the first time you issue a command that needs the GST. After a GST is loaded, it will remain in memory until either the system needs the memory, until you manually unload it using the **gst** utility, or until you tell **sc** to recreate the GST. To load a GST into memory, specify the GST command as follows:

gst *gst-filename*

The **gst** utility loads the GST into memory and then exits.

Note: You should do this only if you plan to remove the disk containing the GST from the drive.

To display information about a symbol contained in a GST, specify the **gst** command as follows:

gst *gst-filename symbol-name*

Some symbols may occur more than once in the GST. For example, a symbol may occur once as a structure tag and again as an identifier. The **gst** utility prints all definitions of the same symbol.

gst supports the following options:

list

lists all GSTs currently in memory. You can use this option to determine which GSTs to unload if you need more memory.

gst Manages global symbol tables
(continued)

[name=]filename

specifies a GST file created using the **makegst** option in the **sc** command. You do not need to specify **name=** unless you want to load a GST that has the same name as a keyword (for example, **name=symbol**).

[symbol=]symbol-name

displays the type of symbol name and the filename and line number where *symbol-name* is defined. The type may be one of the following:

tag	struct , union , or enum name
identifier	variable or function name
typedef	typedef
include	#include filename
preprocessor	preprocessor macro or #define

You do not need to specify **symbol=** unless you want information about a symbol that has the same name as a keyword (**name**, **unload**, and so on).

unload

forces a GST to be unloaded from memory. If you also specify symbol names on the command line, **gst** prints all definitions for those symbols before it unloads the GST.

verbose

prints the full definition of *symbol-name*.

Examples To load the GST named **project.gst**, enter the following command:

```
gst project.gst
```

The **gst** utility loads the GST into memory and then exits.

If the header file **intuition/intuition.h** is included in **project.gst**, you can display the definition of the **Window** structure as follows:

```
gst project.gst Window
```

gst Manages global symbol tables
(continued)

If **project.gst** is not already loaded into memory, **gst** loads **project.gst**. After **project.gst** is loaded into memory, **gst** queries the GST and displays the following information:

```
intuition/intuition.h 904 defines "Window" (struct)
```

To display additional information, enter the following command:

```
gst verbose project.gst Window
```

gst also displays the contents of the **Window** structure with offsets for each member.

Error Messages **gst** may generate the following error messages:

No GST file specified

indicates that you did not specify a GST filename.

Can't find GST file "filename"

indicates that **gst** cannot find the file specified.

Symbol "symbol-name" not found

indicates that **gst** could not find the specified symbol.

Not enough memory to load GST

indicates that the GST file could not be loaded due to insufficient memory.

hypergst Displays the contents of GSTs

Synopsis **hypergst** [*gst-filename*]

Description The **hypergst** utility enables you to browse the definitions of symbols and data structures in GSTs (Global Symbol Tables) that are loaded into memory. The **hypergst** utility works with the AmigaGuide hypertext system. For information on using AmigaGuide, see Chapter 3, “Getting Help,” in *SAS/C Development System User’s Guide, Volume I: Introduction, Editor, Compiler*.

To browse the definitions in a GST, the GST must be loaded in memory. You can invoke the **hypergst** utility and load a GST in one of the following two ways:

- You can enter the **hypergst** command followed by the name of the GST file that you want to browse.
- From the Workbench you can click on the **HyperGST** icon, then hold down the Shift key and double-click on the icon for the GST file that you want to browse.

If the GST that you want to browse is already loaded into memory, you can invoke **hypergst** in one of the following two ways:

- You can enter the **hypergst** command without specifying a GST filename. The **hypergst** utility displays a screen listing the names of all GSTs in memory. To select a specific GST, click on the name of that GST file.
- You can double-click on the **HyperGST** icon. The **hypergst** utility displays a screen listing the names of all GSTs in memory. To select a specific GST, click on the name of that GST file.

After the GST is loaded, **hypergst** displays a screen containing buttons for the various kinds of symbols defined in the GST. These buttons are as follows:

Data Items

displays external data items declared in header files in the GST such as **DOSBase**.

Prototypes

displays prototypes for functions including system functions and functions declared in your own header files.

Typedefs

displays custom data types defined with a **typedef** statement either in system header files or in your own header files.

hypergst Displays the contents of GSTs
(continued)

Structs/Unions/Enums

displays structure, union, and enumerated type definitions.

Preprocessor Symbols

displays preprocessor macros defined in any header file including macros with arguments and simple substitution macros.

If you click on one of these buttons, **hypergst** displays an alphabetical list of all symbols of that type in the GST. If you click on the name of a symbol, **hypergst** displays a symbol information screen for that symbol. This screen may contain other buttons that reference header files, structure tags, type definitions, and so on. You can click on any of these buttons to display more information about the highlighted item.

Error Message **hypergst** may generate the following error message:

No GST files in memory

indicates that you have not loaded any GST files into memory. For example, you see this message if you invoke **hypergst** immediately following a reboot and do not specify any arguments.

lprof Generates run-time statistics

Synopsis `lprof [-t=n] [>prog-output-dest] program [program-options]`

Description The **lprof** utility generates run-time statistics for a program. **lprof** records the amount of time spent in each routine that is called as your program runs. The **lstat** utility produces a report using the information compiled by **lprof**. This report will help you identify modules that run slowly or inefficiently.

To use **lprof** with your program, you must compile your program with the **debug** option.

lprof gathers statistics by determining, at regular intervals, what instruction your program is executing. The default interval is 33 milliseconds. You can change this interval by specifying a new interval (*n*) in milliseconds with the **-t** option as follows:

```
lprof -t=n program
```

You can redirect your program's output by specifying a greater than (>) sign followed by a destination. **program** is the name of the executable that you want to run. If *program* is not in the current directory, you must specify the full pathname. You can specify options for your program.

After **lprof** loads your program, it runs the program and gathers statistics by examining the current program counter at given intervals. **lprof** writes your program's statistics to the file named **prof.out** in the current directory. You can use the **lstat** utility to produce a report from those statistics.

Examples To gather statistics on a simple program named **myprog** that does not take any arguments, enter the following command:

```
lprof myprog
```

For a program that normally takes command-line arguments, you can add them after the program name, as follows:

```
lprof myprog -opt 1 -flag
```

If you want to redirect the program's output to a file named **outfile**, enter the following command:

```
lprof >outfile myprog -opt 1 -flag
```

lprof Generates run-time statistics
(continued)

If the program is not in the current directory, you must give the full pathname, as follows:

```
lprof >outfile C:myprog -opt 1 -flag
```


lstat Analyzes and prints run-time statistics

Synopsis `lstat [>destination] [options] program [profile]`

Description The `lstat` utility analyzes the profile statistics file created by `lprof`. `lprof` runs the program, gathers statistics by examining the current program counter at given intervals, and writes this information to the file named `prof.out` in your current directory. `lstat` analyzes the information in this file and displays a report on the screen.

You can redirect the output from `lstat` by specifying a greater than (>) sign followed by the destination where you want the output sent. You may want to redirect the output to a file and print that file.

The *profile* is the name of the output file created by `lprof`. `lstat` first looks in your current directory for the default file produced by `lprof`, `prof.out`. If this file is not in your current directory, or if you have changed its name, you must specify its full pathname as the value for *profile*.

You must specify the *program* name. `lstat` uses this name to get debugging and symbol information for its report.

Note: This utility assumes that you have compiled the program with the `debug` compiler option. The `debug` option allows `lstat` to associate code with a specific line. If you do not compile with the `debug` option, `lstat` reports statistics based on subroutines only.

`lstat` supports the following options:

- z tells `lstat` to display statistics for all subroutines even if they were not encountered in profiling. By default, `lstat` does not report subroutines that it does not encounter.
- f tells `lstat` to display full statistics for each subroutine, indicating the line numbers within a module that it encountered. By default, `lstat` displays only summary information about the subroutine.
- t=*n* tells `lstat` to display only those routines that have at least *n* hits. By default, `lstat` prints all subroutines that it encountered even once.

If the report produced by `lstat` shows that a large percentage of time is spent in one program module, you may want to

- ☐ redesign the module
- ☐ incorporate the module into the calling routines, thus eliminating calling overhead
- ☐ rewrite the routine in assembly language
- ☐ restructure your program.

lstat Analyzes and prints run-time statistics
(continued)

You also may find that most of the work being done involves the routine in question, and the percentage of time taken is reasonable.

Examples These examples assume that you have the file named **prof.out** in your current directory and that **prof.out** was produced with the following command:

```
lprof testprog
```

To display the basic statistics about your program, enter the command **lstat testprog** as follows:

```
1> lstat testprog
```

```
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
75.367% run, 16.562% ready, 8.071% wait
8.067% samples (58) out of profile range
```

```
counted = 660, RUN - NonTable 661
```

Routine %	Total %	Offset	Hits	Label
59.5455%	59.5455%	1c44c	393	getrsc
8.9394%	68.4848%	1c6fa	59	frersc
3.6364%	72.1212%	10d38	24	alcmem [lines 64-102 in mem.c] Most hits-1.2121% (8) on line 80
2.2727%	74.3939%	18252	15	instal [lines 117-146 in sym.c] Most hits-1.2121% (8) on line 132
...				
0.1515%	100.0000%	1ca18	1	CXM22

You can display information about subroutines that **lprof** did not encounter by using the **-z** option. In this example, using this option produces information about the **strcat** and **strncpy** routines. These routines have 0 hits and account for 0% of the runtime, but now appear in the output, as in the following:

```
1> lstat -z testprog
```

lstat Analyzes and prints run-time statistics
(continued)

```
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
 8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661

Routine %    Total % Offset Hits  Label
59.5455%  59.5455%  1c44c   393  getrsc
 8.9394%  68.4848%  1c6fa    59  frersc
 3.6364%  72.1212%  10d38    24  alcmem [lines 64-102 in mem.c]
                        Most hits-1.2121% (8) on line 80
 2.2727%  74.3939%  18252    15  instal [lines 117-146 in sym.c]
                        Most hits-1.2121% (8) on line 132
...
0.0000% 100.0000%  1ca60     0  strcat
0.0000% 100.0000%  1ca78     0  strncpy
```

For each routine that has line number information, you can display more detailed line number information by specifying the `-f` option. The `-f` option tells `lstat` to display the percentage of total runtime and number of hits for each line that was executed in that routine, as in the following:

```
1> lstat -f testprog

header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
 8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661
```

lstat Analyzes and prints run-time statistics
(continued)

Routine %	Total %	Offset	Hits	Label
59.5455%	59.5455%	1c44c	393	getrsc
8.9394%	68.4848%	1c6fa	59	frersc
3.6364%	72.1212%	10d38	24	alcmem [lines 64-102 in mem.c]
	0.3030%		(2)	on line 64
	1.0606%		(7)	on line 68
	0.7576%		(5)	on line 69
	1.2121%		(8)	on line 80
	0.3030%		(2)	on line 90
2.2727%	74.3939%	18252	15	instal [lines 117-146 in sym.c]
	0.1515%		(1)	on line 117
	0.1515%		(1)	on line 129
	0.4545%		(3)	on line 131
	1.2121%		(8)	on line 132
	0.1515%		(1)	on line 133
	0.1515%		(1)	on line 138
	...			
0.1515%	100.0000%	1ca18	1	CXM22

For a large program, you may want to limit the amount of information produced by `lstat` by using the `-t` option. You can use the `-t` option to set a minimum number of hits that must be recorded for a subroutine before that subroutine will be included in the display. With this option, the final total percentage may not be 100%. Here is an example:

```
1> lstat -f -t=2 testprog
```

```
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
75.367% run, 16.562% ready, 8.071% wait
8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661
```

Istat Analyzes and prints run-time statistics
(continued)

Routine %	Total %	Offset	Hits	Label
59.5455%	59.5455%	1c44c	393	getrsc
8.9394%	68.4848%	1c6fa	59	frersc
3.6364%	72.1212%	10d38	24	alcmem [lines 64-102 in mem.c]
				0.3030% (2) on line 64
				1.0606% (7) on line 68
				0.7576% (5) on line 69
				1.2121% (8) on line 80
				0.3030% (2) on line 90
2.2727%	74.3939%	18252	15	instal [lines 117-146 in sym.c]
				0.1515% (1) on line 117
				0.1515% (1) on line 129
				0.4545% (3) on line 131
				1.2121% (8) on line 132
				0.1515% (1) on line 133
				0.1515% (1) on line 138
				...
0.3030%	98.3333%	1ca00	2	xcovf

omd Disassembles object modules

Synopsis `omd [>destination] [-x] object [source]`

Description The **omd** (Object Module Disassembler) utility program disassembles an object file produced by the SAS/C Compiler and produces a listing consisting of assembly language statements (interspersed with the original C source code if you specified a *source* filename).

omd sends the output to the screen unless you redirect it by entering a greater than (>) sign followed by a *destination*. You may want to send the output to a file and then print the file.

The **-x** option sets the size of the buffer used to hold the external symbol section of the object module. For example, **-x250** establishes a buffer that can hold 250 external symbols. The default size is 200. You should increase the buffer size only if **omd** reports that there are too many external symbols.

object is the object filename. You must specify the complete filename, including the **.o** extension.

source is the source filename. If you specify *source*, you must specify the complete source filename, and the source file should have been compiled with the **debug** compiler option. The **debug** option allows **omd** to associate source lines with the object code they generated. If you did not use the **debug** option, C source lines will not appear in the output produced by **omd**.

Examples The following example compiles **myprog.c** to produce **myprog.o**, which is then disassembled with **omd**. The disassembled listing is redirected to the file named **myprog.lst**.

```
sc debug=line myprog
omd >myprog.lst myprog.o myprog.c
```

oml Manages libraries

Synopsis **oml** [*<cmdfile*] [*>listfile*] [*options*] *libfile* [*command* [*module . . .*]] . . .

Description You can use the **oml** (Object Module Librarian) to manage your libraries. **oml** allows you to:

- ☐ create library files
- ☐ list the modules in a library file
- ☐ delete modules from a library file
- ☐ replace old modules with new modules.

A *link library* is a group of object modules, each of which was originally in a separate file and consisted of one or more subroutines. Some advantages of using a link library are as follows:

- ☐ The subroutines in a library can be used by several programs.
- ☐ When you are linking your program, you specify only the library file instead of several individual object modules.
- ☐ The linker includes only those modules in the library that are required by your program.

Each module within the library is identified by a module name, which is placed in the object module by the program (the SAS/C Assembler or Compiler) that generates the module. The assembler and compiler use the name of the object file.

If a module does not contain a module name, **oml** assigns a module name of the form *\$nnn*, where *nnn* is a decimal number.

A module may define one or more public symbols. A *public symbol* is something in the module that is available to other modules, such as global variables or functions. When the linker resolves external references for a program, it examines each module in each library you specify. It decides which of the modules are required by the program it is linking by looking at the module's list of public symbols. If any of the program's unresolved external references match any of the module's public symbols, that module will be included in the executable module.

When writing the modules for your library, be careful to avoid duplicate names for public symbols. If you create a library that defines a symbol more than once, **oml** issues a warning message. If you then link with that library without correcting the problem, the linker may include the wrong module or give a duplicate symbol error.

To invoke **oml**, enter the **oml** command followed by a library name, as follows:

```
oml mylib.lib
```

oml Manages libraries
(continued)

oml waits for you to enter commands (and module names, if necessary) from the keyboard. **oml** executes the commands as you enter them and displays a list of any duplicate symbol names that it finds. After you have entered all your commands, enter a Control-backslash (Control-\) to terminate the **oml**.

You also can include the commands and module names on the **oml** command line. If you enter commands on the command line, **oml** executes those commands and terminates. For example, the following command replaces the module **ftoc.o** in **mylib.lib** with the version of the same module in your current directory:

```
oml mylib.lib r ftoc.o
```

oml replaces the module, lists any duplicate symbol names it finds, and terminates.

You also can enter some commands on the command line followed by some from the keyboard. If you include an at (@) sign at the end of the command line, **oml** executes the commands you enter on the command line and then waits for you to enter additional commands from the keyboard. You must enter a Control-\ to terminate **oml**. For example, the following command replaces the module **ftoc.o** in **mylib.lib** with the version of the same module in your current directory and then waits for you to enter additional commands from the keyboard:

```
oml mylib.lib r ftoc.o @
```

You also can enter the commands into a file and tell **oml** to use that file as input. You can tell **oml** to use that file by either of the following methods:

- entering a less than (<) sign followed by the filename (**<cmdfile**) as the second item on the **oml** command line
- entering an at (@) sign followed by the filename (**@cmdfile**) as the last item on the **oml** command line.

oml sends its output to the screen unless you redirect it by specifying a greater than (>) sign followed by a filename.

oml Manages libraries
(continued)

Specifying Commands

Commands specify the actions to be performed by **oml** with respect to the specified library file. Each command is specified as a single character, usually followed by a list of module names or object filenames. Separate commands and file and module names with one or more spaces. If you specify the **r**, **d**, or **x** command, **oml** assumes that the next item on the command line is a file or module name, so that item is not checked to determine if it is another command. If you need to enter a file or module name that may appear to **oml** to be a command, specify that file or module name as the first name following the command. **oml** supports the following commands:

r *file file . . .*

replaces the named object files in the library or adds them to the library, if they are not already present. Each module should be in a separate object file. To replace modules within a library, make sure that the module contains a module name and that the filename is the same as the module name.

d *module module . . .*

deletes the named modules from the library. **oml** assigns module names to those modules without names; therefore, you may need to get a listing of names in the library (using the **l** command) to determine the name of the module that you want to delete.

x *module module . . .*

extracts the named modules from the library and creates separate files using the same names. You may need to get a listing of names in the library (using the **l** command) to determine the correct name of the module that you want to extract. If the module name contains a pathname, **oml** attempts to create a file with that name. If the module name does not contain a pathname, **oml** creates the file in the current directory unless you specify a different pathname with the **-o** option. You can use an asterisk (*) to specify that you want all modules extracted. **oml** terminates if it cannot extract a module. You cannot extract and replace the same module with the same **oml** command line.

l

generates a listing of the modules in the library after all other commands have been executed. If you also specify the **-s** option, the listing includes the public symbols defined in each module.

oml Manages libraries
(continued)

a[*filename*]

tells **oml** to execute the commands that you enter from **stdin** (usually defined as the keyboard) or from the *filename*, if specified. If you include this option it should be the last option on the command line.

If you specify the **r** or **d** command, **oml** creates a temporary file in which it builds the new version of the library. After **oml** builds the new version, if it does not detect any errors, it deletes the original library file (if it existed) and renames the temporary file. Therefore, the original library file is not affected if an error occurs.

Specifying Options

oml supports the following options:

- b** tells **oml** not to issue prompts, including prompts for missing information.
- n** strips debugging information from modules before adding them to the library.
- oprefix** specifies a prefix for the filenames to be created by the **x** command. If the prefix includes a directory name, you must include the forward slash (/) at the end of the name.
- s** includes, in the listing produced by the **l** command, a list of the public symbols defined in the module.
- tpathname** specifies a path for the temporary library.
- v** tells **oml** to display messages as it adds or deletes modules to and from the library.
- x** generates a cross reference for variables and functions used in a library.

oml produces warning messages if either of the following are true:

- ☐ A module that you want to delete or extract is not in the library.
- ☐ Any module in the library includes a second definition for a public symbol.

oml Manages libraries
(continued)

Examples Since a single object file is essentially a library of one module, you can use **oml** to find out what module name is included in the object file, as follows:

```
oml name.o l
```

To build a new library, you can create a list of the filenames of the object modules that you want to include in the library and then create the library using the following command:

```
oml new.lib r @name.lst
```

The following command extracts all of the modules in the library **cfuncs.lib** and creates a file for each module in the directory **:object/**:

```
oml -o:object/ cfuncs.lib x *
```

Note: This command will succeed only if the module names in the library do not contain pathnames.

The following command deletes the module **tribe.o** from the library **cfuncs.lib**:

```
oml cfuncs.lib d tribe.o
```

The following command lists the modules and symbols in the library **test.lib**:

```
oml -s test.lib l
```

You can save this listing in a file named **test.lst** by adding the greater than (**>**) sign followed by the filename:

```
oml >test.lst -s test.lib l
```

scmsg Searches for and corrects errors and warnings

Synopsis **scmsg** [*options*]

Description The **scmsg** utility helps you find and fix errors and warnings in your code. It acts as a filter between you and the compiler, allowing you to invoke the editor of your choice and go to the line causing the error or warning quickly and easily.

You can use **scmsg** with the editor of your choice, using AREXX as a communication medium. AREXX is a standard part of AmigaDOS 2.0 and can be purchased as a third-party product for use under AmigaDOS 1.3. If you do not have AREXX, you still can use **scmsg** to control the **se** editor provided as part of the SAS/C Development System, but you will not be able to program the editor keys to control **scmsg**.

If you invoke the compiler from the Shell or by using the **Build** icon on the Workbench, **scmsg** is automatically invoked for you.

You can specify the following options in the **scmsg** command:

autoedit

specifies that you want **scmsg** to invoke your editor automatically, open the source file, and display the line producing the message. The default value is **noautoedit**.

config=filename

specifies the name of a configuration file. Typically, the configuration file contains the command necessary to invoke your editor, open a source file, and display the appropriate line. For more information about configuration files, see “Using AREXX to Invoke Your Editor,” later in this section.

hidden

specifies that you do not want **scmsg** to display the message browser window until the compiler returns a message. The default value is **nohidden**.

To redisplay the message browser window, you can enter **scmsg nohidden** at the Shell prompt or send the AREXX **show** command to the **SC_SCMSG** AREXX port. For example, you can use the following AREXX script to send the **show** command:

```
/* AREXX script to send the SHOW command */
ADDRESS 'SC_SCMSG'
'SHOW'
```

quit

terminates **scmsg**.

scmsg Searches for and corrects errors and warnings
(continued)

rexonly

specifies that **scmsg** should not open a window. Use this option if you intend to query **scmsg** for the messages from your editor or some other AREXX-supporting program, and you do not want to see the **scmsg** window.

Unless you specify **hidden** or **rexonly**, **scmsg** opens a window on the Workbench screen. This window contains the first error or warning message from the compilation. As additional messages are produced, they are appended to the end of the list. These additional messages may be produced in the same or a different source file.

The *current message* is always highlighted. You can move from message to message by clicking on the message or by using the arrow keys. The scroll bar on the right side of the window allows you to move around in the list. Double-clicking on a message invokes the editor at the file and line number specified in the message.

The **scmsg** window lists all messages in the following format:

```
primary: class msgno in secondary line lineno: text
```

where:

primary

is the C source file being compiled.

class

is either **Error** or **Warning**.

msgno

is the message number.

secondary

is the name of the file in which the error or warning occurred. This file may be the same file as the primary file, or it may be a header file included by the primary file using the **#include** statement.

lineno

is the line number in the secondary file.

text

is the message text. The message text may specify an *alternate file* for a message. If a message describes a conflict between two different places in your code, the alternate file gives the location of one of them. The secondary file gives the location of the other. If an

scmsg Searches for and corrects errors and warnings
(continued)

alternate file is available for the message, the text contains the words

See line *num* file *filename*.

Every time a C source file is compiled, all old messages listing that file as the primary file are deleted from the **scmsg** window.

The following sections describe the options available from the Project and Edit menus on the **scmsg** screen. Many of these menu items have *menu accelerator keys* that allow you to perform the specified action without selecting the menu item. To use a menu accelerator key, hold down the right Amiga key (just to the right of the spacebar) and press the specified key.

The Project Menu

The Project menu allows you to control your message session. The following list describes each option on the Project menu and, where applicable, gives the menu accelerator key equivalent.

Set Options

sets **scmsg** options that specify which editor you want to use and any options for that editor. These options are typically saved in the file **ENV:sc/scmsg**. The options available are as follows:

editcommand

is the Shell command necessary to start your editor. Use the **%f** sequence to indicate where the filename should be in the command. If this option is left blank, **scmsg** does not attempt to invoke your editor, and all the other options are ignored. If this option is left blank but a portname is specified, **scmsg** sends AREXX commands to the port, if it exists.

gotofile

specifies the AREXX command template needed to open a file. Use the **%f** sequence to indicate where the filename should be in the command. Once the filename has been substituted, this command is sent to your editor's AREXX port when a new file needs to be opened. If this option is left blank, **scmsg** always uses the **editcommand** to invoke your editor each time there is an error.

scmsg Searches for and corrects errors and warnings
(continued)

gotoline

specifies the AREXX command template needed to go to a specific line. Use the %1 sequence to indicate where the line number should be in the command. Once the line number has been substituted, this command is sent to your editor's AREXX port when **scmsg** needs to go to a new line. If this option is left blank, **scmsg** does not attempt to move to the proper line number in the file.

hidden

specifies that **scmsg** should not open a window. Use this option if you intend to query **scmsg** for the messages from your editor or some other AREXX-supporting program and you do not want to see the **scmsg** window. This option does not take any arguments.

portname

specifies the name of your editor's AREXX port. If this option is left blank, but **editcommand** is specified, **scmsg** assumes that your editor does not support AREXX. The **scmsg** utility issues the **editcommand** option but does not attempt to send any AREXX commands to your editor.

wait

specifies that **scmsg** should not open a window until one or more new compiler messages are available. This option does not take any arguments.

Use

saves the current options to **ENV:sc/scmsg**.

Save

saves the current options in **ENV:sc/scmsg** and **ENVARC:sc/scmsg**. The **left**, **top**, **width**, and **height** options are set to the current window's values.

Save As . . .

allows you to save your current options to a specified filename. You will be prompted for the filename.

Restore

reads the option settings from **ENV:sc/scmsg** into memory.

Restore from . . .

reads the options settings from a specified filename. The **scmsg** utility prompts you to enter the filename.

scmsg Searches for and corrects errors and warnings
(continued)

Reset to SAS/C defaults

resets all options to the default values provided by the SAS/C Development System. Refer to Chapter 8, “Compiling and Linking Your Program,” in *SAS/C Development System User’s Guide, Volume I: Introduction, Compiler, Editor*. These defaults are suitable for use with **se**.

Hide Window

closes the **scmsg** window until the compiler produces another message.

To redisplay the message browser window, you can enter **scmsg nohidden** at the Shell prompt or send the AREXX **show** command to the **SC_SCMSG** AREXX port. For example, you can use the following AREXX script to send the **show** command:

```
/* AREXX script to send the SHOW command */
ADDRESS 'SC_SCMSG'
'SHOW'
```

Quit

terminates **scmsg**.

The Build Menu

You can use the Build menu to terminate any current builds or start new builds. The following list describes each option on the Build menu and, where applicable, gives the menu accelerator key equivalent.

Abort Build

terminates any current builds that are active. You may want to select this option if, for example, an error in a common header file is producing too many warnings or errors. The menu accelerator key for this command is Right-Amiga-A.

Build Project

starts a new build. The **smake** utility is invoked from the directory specified in the **ENV:sc/projdir** environment variable. This variable is automatically set for you each time you submit a **smake** command or click on the **Build** icon from Workbench. If unset, the current directory is used. The menu accelerator key for this command is Right-Amiga-B. Any build that is currently running is aborted before the new build is submitted.

scmsg Searches for and corrects errors and warnings
(continued)

The Edit Menu

You can use the Edit menu to perform certain actions on the current (highlighted) message. The following list describes each option on the Edit menu and, where applicable, gives the menu accelerator key equivalent.

Go to Error

invokes the editor on the file and line number that produced the message. You can perform the same action by double-clicking on the message or by pressing the Return key.

Go to Alternate

invokes the editor on the alternate file and line. This option is valid for messages that have an alternate file and line number specified in the form

See line *num* file *filename*.

You also can edit the alternate file by holding down either Alt key and double-clicking on the message or by holding down either Alt key and pressing the Return key.

Delete Message

removes the highlighted message from the list. The menu accelerator key is Right-Amiga-D.

Clear

removes all messages from the list. The menu accelerator key is Right-Amiga-C.

Delete by . . .

displays a submenu that allows you to delete all messages that meet specific criteria. The options on the submenu are as follows:

Message Number

deletes all messages with the same message number as the current message. The menu accelerator key is Right-Amiga-N.

File

deletes all messages produced by the same secondary file, regardless of the primary file. The menu accelerator key is Right-Amiga-F.

Compilation

deletes all messages produced by the same primary file. The menu accelerator key is Right-Amiga-P.

scmsg Searches for and corrects errors and warnings
(continued)

Using AREXX to Invoke Your Editor

When **scmsg** wants to communicate with your editor, it issues an AREXX command based on the templates you provide with the Set Options submenu and the information from the highlighted message. It uses the template in the appropriate option as a base and performs substitutions on it. All templates start with a percent (%) sign. The following substitutions are performed:

Template	Substitution
%%	percent (%) sign
%n	newline character
%f	filename
%l	line number
%c	column number
%xhh	hexadecimal character specified by hh
%r	carriage return

scmsg ignores any other sequence of a percent (%) sign followed by a character.

If **scmsg** finds the AREXX port described in your options file, it sends the **openfile** AREXX sequence provided to the port to tell your editor to open the necessary file.

If **scmsg** cannot locate the AREXX port described in your options file, it attempts to invoke your editor on the appropriate file using the specified **editcommand** Shell command. If successful, it goes to the appropriate line using the AREXX command template.

The following values are typical of what might be found in a configuration file created for **se** by the Set Options submenu under the Project menu:

editcommand	SC:C/SE	Shell command to invoke SE
portname	SC_SE	Name of the editor's REXX port

scmsg Searches for and corrects errors and warnings
(continued)

gotofile	OW "%f"%r	Open specified file
gotoline	LL "%l"%r	Go to line number %l.

Configuration files and example AREXX scripts for several popular editors have been included with the SAS/C Development System in the **sc:extras/scmsg** drawer. Check that drawer to see if support for your editor has been provided.

Using the **SC_SCMSG** AREXX Port

You can also use the built-in AREXX port in **scmsg** to get information on the messages remotely from your editor or from any other program that supports AREXX. The port's name (for use with the AREXX **address** command) is **SC_SCMSG**. The following commands are supported:

abort

aborts any builds currently running.

altfile

returns the alternate filename (if any). An empty string indicates that there are no messages on the list or that the current message has no alternate file.

altline

returns the alternate line number (if any). An empty string indicates that there are no messages on the list or that the current message has no alternate file.

bottom

goes to the last message in the list.

build

submits a new build of the last project built. Any build that is currently running is aborted before the new build is submitted.

class

returns either **error** or **warning**, depending on the current message.

clear

deletes all messages.

delcomp [filename]

deletes all messages with the specified filename as their primary filename. If no filename is specified, the primary filename of the

scmsg Searches for and corrects errors and warnings
(continued)

current message is used. If the current message is deleted, the next non-deleted message becomes current.

delete

deletes the current message and goes to the next message in the list.

del file [filename]

deletes all messages with the specified filename as their secondary filename. If no filename is specified, the secondary filename of the current message is used. If the current message is deleted, the next non-deleted message becomes current.

del num [msgno]

deletes all messages with the specified message number. If no message number is specified, the number of the current message is used. If the current message is deleted, the next non-deleted message becomes current.

errnum

returns the error number. An empty string indicates that there are no messages on the list.

file

returns the filename for the current message. An empty string indicates there are no messages on the list.

hide

closes the **scmsg** window, but keeps **scmsg** running. If you specify **wait**, the **scmsg** window reappears when a new message is issued from the compiler. Otherwise, you must send a **show** command with REXX or reinvoke **scmsg** from the command line to redisplay the window.

line

returns the line number for the current message. An empty string indicates there are no messages on the list.

next

goes to the next message in the list. If the current message is the last in the list, **scmsg** goes to the first message in the list.

prev

goes to the previous message in the list. If the current message is the first in the list, **scmsg** goes to the last message in the list.

quit

terminates **scmsg**.

scmsg Searches for and corrects errors and warnings
(continued)

rexonly

specifies that **scmsg** should close the message browser window and not reopen it unless you specify **scmsg nohidden** at the Shell prompt or send the ARGXX **show** command to the **SC_SCMSG** ARGXX port. Use this command if you intend to query **scmsg** for the messages from your editor or some other AREXX-supporting program, and you do not want to see the **scmsg** window.

select

selects the current message for editing. Your editor is invoked and moved to the proper file and line number. This command is equivalent to pressing the Return key on the current message.

show [activate]

pops the **scmsg** window to the front or redisplay the window if it is hidden. If you specify **activate**, the **scmsg** window becomes the active window, and you can enter keyboard commands to **scmsg**.

text

returns the message text. An empty string indicates that there are no messages on the list.

top

goes to the first message in the list.

Examples The following AREXX script, when invoked from **se**, opens the file containing the current message, moves to the appropriate line number, and displays the message on the editor message line:

```
/* AREXX script to move to next error */
address 'SC_SCMSG' /* Set up to talk to scmsg */
'delete'          /* Delete the current error, move to next */

options results   /* Tell AREXX we want the return values */
                  /* of issued AREXX commands           */

'file'            /* Get the new filename */
file = result

'line'            /* Get the new line number */
line = result
```

scmsg Searches for and corrects errors and warnings
(continued)

```
'text'                /* Get the new message text */
text = result

options                /* Don't care about results any more */

address 'SC_SE'        /* Now talk to the editor */

if file = "" then
    'DMNo more errors' /* Display Message "No more errors" */
else do
    'OW UC' || file || "0d"x /* Open Window "file"      */
    'LL UC' || line || "0d"x /* Go to line "line"    */
    'DM' || text           /* Display Message "text" */
end
```

scopts Sets compiler options

Synopsis **scopts** [*options*]

Description The **scopts** utility allows you to set compiler options for a project by clicking on the gadget that corresponds to the option. The options you specify are used by the **sc** command whenever you compile and link your files.

To run **scopts**, you can double-click on the **SOptions** icon, or you can enter **scopts** on the Shell command line. **scopts** displays the SAS/C Compiler Options Index window. This window contains buttons that open additional windows. Each of these windows allows you to set different compiler options. In all, **scopts** can display nine windows:

Compiler Options Index

allows you to do the following:

- ☐ set certain options such as **list**, **xref**, **link**, and **map**
- ☐ specify the name of the final executable module
- ☐ display additional windows such as the Optimizer Options window and the Link Options windows
- ☐ save your option settings to a file named **scoptions** in the current directory
- ☐ save your option settings in **ENV:sc/scoptions**.

Compiler Options

contains gadgets for miscellaneous options such as **debug**, **shortintegers**, and **stringmerge**.

Diagnostic Message Options

contains gadgets for options, such as **ansi**, **strict**, and **errorrexx**, that affect which messages are generated by the compiler and linker and how those messages are handled.

Code Generation Options

contains gadgets for options, such as **data**, **code**, and **cpu**, that affect the code generated by the compiler.

Listing/Cross-Reference Options

contains gadgets for options, such as **listnarrow**, **errorlisting**, and **xrefsystem**, that affect the content and formatting of any listing or cross-reference files generated.

Optimizer Options

contains gadgets for options, such as **optpeep**, **optsize**, and **optalias**, that affect the performance of the optimizer.

scopts Sets compiler options
(continued)

Prototype Generation Options

contains gadgets for options, such as **genprotoexterns**, **genprotoparms**, and **genprototypedefs**, that affect how prototypes are generated.

Linker Options

contains gadgets for options, such as **smallcode**, **stripdebug**, and **batch**, that are passed to the linker when the linker is invoked by **sc**. These options are not used by the compiler.

Map Options

contains gadgets for options, such as **maphunk**, **mapxref**, and **mapoverlay**, that affect the content of the map file.

For a brief description of an option, move the cursor to the option gadget and press the Help key. For a complete description of any compiler option, refer to Chapter 8, “Compiling and Linking Your Program,” in *User’s Guide, Volume I*.

These windows may contain one or more of the following basic types of gadgets:

- cycles appear as raised buttons with a cycle symbol on the left side. By clicking on the gadget with the left mouse button, you can cycle through the setting available for that option. To reverse the direction of the cycle, press the Shift key as you click the mouse button.
- actions appear as raised buttons. Selecting the button causes an immediate action to occur. For example, selecting the **Compiler Options...** gadget displays the Compiler Options window.
- strings appear as a raised ridge around a text area. You can enter data in the area by clicking within the text area and typing. When you have finished typing, press the Return key.
- lists appear as a set of control gadgets (a scroll bar, arrow gadgets, a string area, and **ADD** and **DEL** buttons) combined with a raised scrolling area. Use the scroll bar and arrow gadgets to position the scrolling area on specific items. To add new items to the list, select the **ADD** button, type the new item name, and press the Return key. To remove

scopts Sets compiler options
(continued)

an item from the list, click on the item (to copy it to the string area), and select the **DEL** button. To modify an item, click on the item (to copy it), and enter any corrections.

To specify an option for which the **scopts** utility does not have a gadget, enter the option and any parameters required by the option into the **SPECIAL** gadget. You can enter as many additional options as you like. If you use the **SPECIAL** gadget to specify an option for which **scopts** already has a gadget, the next time you invoke **scopts**, the gadget for that option is selected, and the option is no longer included in the **SPECIAL** gadget. The only options remaining in the **SPECIAL** gadget are those for which **scopts** does not have a gadget. For example, if you enter **link** into the **SPECIAL** gadget, the next time you invoke **scopts**, the **link** gadget is selected, and the **SPECIAL** gadget does not contain the **link** option.

Any values you enter into the **SPECIAL** gadget are processed after all other gadgets are processed. Therefore, options you enter into the **SPECIAL** gadget may override other options.

To specify the name of your final executable module, specify a filename in the **ProgramName** gadget.

To save your option settings, select one of the following options from the Project menu:

Save to SCOPTIONS

saves the settings to the file **scoptions** in the current directory.

You also can click on the **Save** button in the SAS/C Compiler Options Index window to save the settings in **scoptions**.

Save as global defaults (ENV:)

saves the settings to the file **ENV:/sc/scoptions**. You also can click on the **Save Default** button in the SAS/C Compiler Options Index window to save the settings in **ENV:.** The option settings are also saved to **ENVARC:**, if it exists.

Save as . . .

asks you for the name of the file in which you want to save the settings. To use the options saved in a file other than **scoptions** or **ENV:sc/scoptions**, you can specify the filename using the **with** compiler option in the **sc** command. The options in the **with** file are read as if they were specified in the **sc** command at the position occupied by the **with** option.

scopts Sets compiler options
(continued)

To exit **scopts** without saving any of your changes, click on the **Cancel** button.

When you run the **sc** command, it first looks for an **scoptions** file in your current directory. If this file does not exist, the **sc** command looks for the **ENV:sc/scoptions** file.

The **scoptions** file is an ASCII file that contains the list of **sc** options that you specify. You can edit this file with a text editor and add any options you want.

To load a previously saved options file or to reset all options to their default values, select one of the following options from the Project menu:

Restore from SCOPTIONS

reads the contents of the **scoptions** in the current directory into memory.

Restore from global defaults (ENV:)

reads the contents of **ENV:sc/scoptions** into memory.

Restore from . . .

asks you for the name of the file from which you want to read option settings into memory.

Reset to SAS/C defaults

resets all options to the default values provided by the SAS/C Development System. Refer to Chapter 8, "Compiling and Linking Your Program," in *SAS/C Development System User's Guide, Volume I: Introduction, Compiler, Editor*.

You also can change option settings by specifying the option on the **scopts** command line. For example, to set the **math=standard** and **link** options, you can enter the **scopts** command as follows:

```
scopts math=standard link
```

If you enter the **scopts** command followed by option settings, **scopts** does not display any windows, and you cannot choose the file into which the options are saved. Specifically, **scopts** performs the following tasks:

1. reads the **scoptions** file in your current directory, if it exists.
If this file does not exist, **scopts** reads **ENV:sc/scoptions**.
2. adds or changes the options you specified.

scopts Sets compiler options
(*continued*)

3. saves the new option settings in the file **scoptions** in the current directory.

scsetup Sets up a new project

Synopsis `sc:scsetup [directory] . . .`

Description The **scsetup** utility sets up a directory that you can use for C development. You can run **scsetup** on existing directories or use it to create new directories. **scsetup** creates an icon for the directory and copies into the directory icons for the most commonly used SAS/C Development System tools, such as **smake** and CodeProbe. You can add icons for any file extensions or tools that you want **scsetup** to copy in addition to the standard icons.

Specifically, **scsetup** performs the following actions:

1. Creates the directory, if necessary.
2. Creates an icon file for the directory being set up, if it does not already have one. **scsetup** uses the file `sc:starter_project.info` as a template for directory icons.
3. Creates icons for the files in the directory, if necessary. Default icons are supplied for `.c`, `.h`, `.a`, and `.smk` files. These icons are in the `sc:icons` drawer under the name `def_c.info`, `def_h.info`, `def_a.info`, and `def_smk.info`. You can add any default icons needed for other extensions by copying an icon into `sc:icons` with the appropriate name (`def_extension.info`).
4. Creates an icon file for any subdirectories of the directory being set up, if they do not already have one.
5. Creates icons for the files in each subdirectory, if necessary.
6. Copies the contents of the directory `sc:starter_project`, if present, to the directory being set up. This starter directory contains a **Build** icon, a **Debug** icon, an **Edit** icon, and an **SCOptions** icon. You can place any other icons or programs in this starter directory that you want copied to directories set up with **scsetup**.

scsetup will not overwrite any existing files or icons.

You can run **scsetup** from the CLI or from the Workbench. To run **scsetup** from the CLI, enter the following command:

`sc:scsetup [directory] . . .`

If you do not specify a directory name, **scsetup** sets up the current directory. If you specify a directory that does not exist, **scsetup** creates the directory.

To run **scsetup** from the Workbench and create a new project, double-click on the **SCSetup** icon, and type the name of the drawer when prompted.

scsetup Sets up a new project
(continued)

To run **scsetup** from the Workbench and set up an existing drawer, click on the drawer icon. (Click on the actual drawer icon, not the file icons in the drawer.) To select additional drawers, hold down the Shift key and click on their icons. After you have selected all the drawers, hold down the Shift key and double-click on the **SCsetup** icon.

After you have set up a directory, you can run the editor by double-clicking on the **Edit** icon, and you can run CodeProbe by double-clicking on the **Debug** icon. You can run **smake** by double-clicking on the **Build** icon. **smake** looks for a smakefile or a makefile. If none are present, it compiles and links all **.c** files in the directory using the options specified through the **scopts** utility, described earlier in this chapter.

smake Maintains and updates records of file dependencies

Synopsis **smake** [*options*] [*macro-definitions*] [*targets*]

Description The **smake** utility is a tool that you can use to maintain projects that are composed of many files. A file can be C source code, a data file for a graphics or audio, or perhaps a spreadsheet file. For example, you may have a project that consists of 50 files, and several programmers may be responsible for different files. To manage this project, you need to keep track of which files have been changed and which files must be compiled before other files (that is, which files are dependent on other files). You can use **smake** to keep track of file dependencies, recompile and relink any files that have been updated, and produce new product files.

In other words, **smake** determines if any of the source files have changed since the last version of the product file was generated and, if so, generates a new product file.

To use **smake**, you create a *smakefile* that identifies *dependent files* and *target files* and describes the *actions* required to produce a new product. A basic description of these terms follows:

- target file is any file that is created or updated by **smake**. Producing this file may require creating or updating several intermediate files.
- dependent file is a file that must be created or updated before a target file can be updated. A dependent file can be a source code file, header file, or object code file. In other words, if a dependent file is changed (for example, by editing), then the target file must be rebuilt.
- actions are the commands necessary to update each dependent and target file. Actions can be calls to the compiler or linker or basic housekeeping commands.
- smakefile is a file that identifies every dependent file required to create the target file and describes every action required to update or create the dependent and target files.

When you run **smake**, it re-creates only the first target file specified in the *smakefile* (unless you specify an alternate target as described in “Using Alternate Targets,” later in this section) and, if necessary, any

smake Maintains and updates records of file dependencies
(continued)

dependent files needed to re-create the target file. Specifically, **smake** performs the following actions:

1. locates and identifies the target file. By default, **smake** remakes the first target file specified in the smakefile. You can specify an alternate target as described under “Using Alternate Targets,” later in this section.
2. ensures that any file on which the target file depends already exists and is up to date. To determine when dependent files were last modified, **smake** looks at the time stamp of the file. A file is considered *current* if it has a time stamp later than that of any of its dependent files.

Note: Before using **smake**, make sure that your system clock is accurate, using the **date** command if appropriate.

An example of the date command under AmigaDOS follows:

```
date 12-Aug-92 16:34:57
```

You must enter the time using 24-hour clock notation. Enter the month as a three-character field.

3. Re-creates the target file if any of the dependent files have been modified more recently than the target file.

Creating the smakefile A smakefile has the following basic format:

```
# This is a comment.
target-file1: dependent-file1 dependent-file2...
    actions
target-file2: dependent-file1 dependent-file2...
    actions
```

You can include comments in a smakefile by entering a pound (#) sign in the first column. **smake** will ignore the remainder of that line.

All target file names must

- ☐ start in the first column
- ☐ be followed by a colon (:)
- ☐ be followed by the name of any dependencies of the target.

smake Maintains and updates records of file dependencies
(continued)

If there are too many dependent filenames to fit on one line, repeat the target name and colon on the next line, and enter the remaining dependent filenames.

Immediately after the target-dependency line are any actions required to generate the target. Each of the action lines, if any, must be indented a minimum of one tab or space. To continue an action line onto additional lines, end each continued line with a backslash (\). You can have as many action lines as necessary for each target-dependency line. You can call **smake** from within a smakefile. You cannot issue the **cd** (change directory) command from within a smakefile.

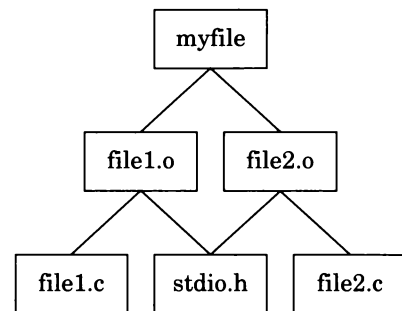
For example, suppose you have an executable file named **myfile** that you created by linking two object files, **file1.o** and **file2.o**. These object files are created by compiling the source files **file1.c** and **file2.c**, and each of these source files includes the header **stdio.h**. Figure 10.1 shows the relationships among these files.

Figure 10.1
Example File
Dependencies

Executable File
(Primary Target)

Object Files
(Dependents of Primary and
Targets of Source Files)

Source Files
(Dependents)



If any information in **stdio.h** changes, you would have to recompile both source files and relink both object files to produce a new executable file. However, you can create a smakefile that contains all of the information necessary to recompile and relink these files and then run **smake** to recompile and relink as necessary.

smake Maintains and updates records of file dependencies
(continued)

Your smakefile could look like this:

```

1  # Primary target is myfile
2  myfile: file1.o file2.o
3      slink FROM LIB:c.o+file1.o+file2.o TO myfile LIB \
4          LIB:sc.lib+LIB:amiga.lib
5
6  file1.o: df1:source/file1.c stdio.h
7      sc data=absolute code=far df1:source/file1.c
8
9  file2.o: df1:source/file2.c stdio.h
10     sc data=absolute code=far df1:source/file2.c

```

Note: Do not enter line numbers in your smakefiles. Line numbers are included here for use in the discussion that follows.

The first line is a comment line. The second line identifies the two dependent object files of the target file **myfile**. That is, it identifies the files that must be current before **myfile** can be generated: **file1.o** and **file2.o**.

Lines three and four give the commands (actions) necessary to produce **myfile**. In this example, the only command necessary to produce **myfile** is a call to the linker, **slink**.

The sixth line identifies the files required to produce the object file **file1.o**. The two files are **df1:source/file1.c** and **stdio.h**. Line seven gives the command necessary to produce the object file: a call to the compiler, **sc**. Lines nine and ten identify the dependent files and actions required to produce **file2.o**.

Target-dependency and action lines are the primary components of a smakefile. However, **smake** provides many special features that you can use to make your smakefiles more powerful. These features include macros, transformation rules, local input files, fake targets, and more. The following sections describe all of the special features you can use in your smakefile.

smake Maintains and updates records of file dependencies
(continued)

Using Special Symbols

You can use special symbols to override some of the default actions of **smake**. These special characters are as follows:

- Normally, when an action generates an error, **smake** stops processing. If you want **smake** to continue processing when it encounters an error caused by a specific action, begin the action line with a minus (–) sign. If you want **smake** to continue processing after all errors, specify the command line option **-k**.
- @ Normally, **smake** displays all commands and messages generated by those commands on the screen as each command is executed. If you do not want a command to be echoed to the screen, begin the action line with the at (@) symbol. **smake** will still display any generated messages. If you do not want any commands echoed to the screen, specify the command line option **-s**.
- \$ The dollar (\$) sign is an escape symbol for the dollar sign itself. In other words, to enter a dollar sign in a filename, enter two dollar signs (\$\$). If you enter only one dollar sign, **smake** thinks that the dollar sign indicates the beginning of a macro.
- \ The backslash (\) symbol is an escape symbol for the pound (#) sign. Pound signs normally indicate the beginning of a comment. To use a pound sign in a command line, enter a backslash before the pound sign (\#).
- & The ampersand (&) symbol is provided for compatibility with other implementations of **smake**. The ampersand is ignored under AmigaDOS.

Using Macros

You can use macros to represent any character string, including AmigaDOS commands, directory specifications, header files, compiler or assembler flags, and constant strings. Macros are especially useful where long character strings are required.

Within a smakefile, you can define a macro using the equals (=) sign as follows:

macro-name = definition

Make sure you define a macro before you attempt to use it. If you reference a macro that has not yet been defined, that macro will be expanded to an empty string.

smake Maintains and updates records of file dependencies
(continued)

Note: You purposely can define a macro to be the empty string by entering only the macro name and the equals (=) sign:

macro-name =

To refer to a macro within a smakefile, use the following format:

\$(macro-name)

For example, you can define the macros **source** and **flags** to represent the pathname of your source file and the compiler options you want to use to compile that file. Using these two macros, our earlier example now looks like the following:

```
# Define macros for source pathname and compiler options
SOURCE = df1:source/
FLAGS = data=absolute code=far

# Primary target is myfile
myfile: file1.o file2.o
    slink FROM LIB:c.o+file1.o+file2.o TO myfile LIB \
        LIB:sc.lib+LIB:amiga.lib

file1.o: $(SOURCE)file1.c stdio.h
    sc $(FLAGS) $(SOURCE)file1.c

file2.o: $(SOURCE)file2.c stdio.h
    sc $(FLAGS) $(SOURCE)file2.c
```

Overriding a Macro Definition

You can override the definition of a macro from the command line of **smake** by specifying the new definition as follows:

macro-name=new-definition

If you specify a macro definition on the **smake** command line, do not include spaces around the equals (=) sign. For example, you can redefine the **debug** macro as follows:

```
smake -f editor debug=yes
```

smake Maintains and updates records of file dependencies
(continued)

Using Default Macros

smake provides five default macros that are defined after a target-dependency line is processed and before the next action line is processed.

Macro	Definition
\$@	target filename
\$?	list of dependent files, including the full directory pathnames, that are not current (that is, they have time stamps later than the target time stamps)
\$*	name of the first dependent file, including the full directory pathname but excluding the filename extension
\$<	name of the dependent file that caused the action, excluding the directory pathname but including the filename extension
\$>	name of the dependent file that caused the action, excluding directory pathnames and filename extension

For example, your smakefile could contain the following target-dependency line:

```
obj/new/test.o: src/new/test.c hdr/test.h
```

This line defines the source file as **test.c** located in the directory **src/new** and the header file **test.h** located in the directory **hdr**. If the source file **test.c** has a time stamp later than the target file **test.o**, then the default macros will have the following values:

Macro	Value
\$@	obj/new/test.o
\$?	src/new/test.c
\$*	src/new/test
\$<	test.c
\$>	test

smake Maintains and updates records of file dependencies
(continued)

Note: The value of each of these default macros changes after each target-dependency line is processed.

Defining Transformation Rules

You can use transformation rules with macros to tell **smake** how to make a file with a given filename extension from another file with a different filename extension. Transformation rules are most useful when you have several files with which you want to perform the same action. **smake** uses transformation rules if you do not specify explicitly the action required to create a certain target.

For example, you can have a transformation rule that tells **smake** how to create object files with extensions of `.o` from source files with extensions of `.c`. If the only action required to produce the object file from the source file is a call to the compiler, your smakefile can contain the following transformation rule:

```
.c.o:  
    sc $*
```

smake Maintains and updates records of file dependencies
(continued)

Then, **smake** would compile source files (**os.c** and **strbpl.c**) without an action line for each source file that explicitly specified the call to the compiler:

```
.c.o:
    sc $*

obj/os.o: os.c

obj/strbpl.o: strbpl.c
```

Using Fake Targets

Fake targets are a set of predefined targets that allow you to control the behavior of **smake**. **smake** does not remake fake targets. Fake targets allow you to set logical names, specify actions to be taken for certain conditions, and suppress certain messages.

smake provides five fake targets:

```
.DEFAULT
.IGNORE
.ONERROR
.SET
.SILENT
```

The following list describes these targets in detail

.DEFAULT

specifies an individual action or a set of actions to be executed if

- ☐ there are no actions specified for a given target-dependency line
- ☐ there is no default transformation rule for the given target-dependency pair.

The command line syntax used for the **.DEFAULT** target is as follows:

```
.DEFAULT:
    action-1
    action-2
    .
    .
    .
    action-n
```

smake Maintains and updates records of file dependencies
(continued)

You can use **.DEFAULT** actions to display messages to the screen as **smake** is running. For example, your smakefile can consist of the following lines:

```
.DEFAULT:
    ; this target needs remaking $@
alpha.cpp: alpha.clp
```

If **alpha.clp** has a later time stamp than **alpha.cpp**, **smake** will print the following message to your screen:

```
this target needs remaking alpha.cpp
```

.IGNORE

tells **smake** to ignore any error caused by the action lines. Specifying **.IGNORE** is equivalent to calling **smake** with the **-i** option. To specify **.IGNORE**, include the following line in your smakefile:

```
.IGNORE:
```

.ONERROR

specifies an action or set of actions to be performed when **smake** detects an error. For example, you may want to delete certain temporary files or transfer them to a different directory if an error occurs. Specify the actions to be taken after the **.ONERROR** target:

```
.ONERROR
    action-1
    action-2
    .
    .
    .
    action-n
```

.SET

allows you to set the values of logical name assignments (as created by the **assign** command) from within a smakefile. When the logical name is set from within a smakefile, the logical name retains

smake Maintains and updates records of file dependencies
(continued)

the specified value and will be in effect for any secondary processes spawned from the main process. To specify **.SET**, use the following format:

```
.SET:  logical-name = value
```

The value may be any character, including a space. However, the leading and trailing space characters are ignored.

For example, you can redefine the logical name **INCLUDE** as follows:

```
.SET:  INCLUDE: = df0:sc/examples
```

.SILENT

tells **smake** not to echo the action lines to the standard output device before they are executed. Specifying **.SILENT** is equivalent to calling **smake** with the **-s** option. To specify **.SILENT**, include the following line in your smakefile:

```
.SILENT:
```

Using Multiple Targets

If you have several target files that are dependent on the same dependent files, you can specify the target files together on the same target-dependency line. For example, if **prog.c**, **myfile.c**, and **newcode.c** all included the header file **stdio.h**, you can use the following lines in your smakefile:

```
prog.c myfile.c newcode.c: stdio.h
sc $@
```

Using Alternate Targets

Unless you specify an alternate target, **smake** remakes only the first target file specified in the smakefile and, if necessary, any dependent files needed to re-create the target file. However, your smakefile can contain target-dependency and actions lines for several targets. To tell

smake Maintains and updates records of file dependencies
(continued)

smake to remake a target other than the first one specified in the smakefile, enter the name of the target on the **smake** command line:

smake *target-name*

For example, you can have the following smakefile that specifies two targets, **clean** and **backup**:

```
clean:
    delete $(OBJ)files.o.DS
    delete $(OBJ)os.o
    delete $(OBJ)strbpl.o

backup:
    copy files.c to df1:
    copy os.c to df1:
    copy strbpl.c to df1:
```

If you enter the **smake** command without specifying a target, **smake** runs the commands listed under **clean**. To run the commands listed under **backup**, you must use this command line:

smake **backup**

Using Local Input Files

From within a smakefile, you can create a temporary disk file containing instructions that is passed to AmigaDOS when you run **smake**. Local input files are useful when you need to enter command lines that are too long to fit on one line.

Any macros defined in the smakefile can be used within a local input file. Local input files are automatically deleted before **smake** terminates.

You can construct a local input file with action lines as follows:

```
command <[preface] <[!][filename)]
    statement1
    statement2
    .
    .
    .
    statementn
    <
```

smake Maintains and updates records of file dependencies
(continued)

where:

- command* is the command to be handed to AmigaDOS.
- preface* is any text or symbol to be passed to AmigaDOS before the temporary filename. The preface may contain spaces.
- !** tells **smake** to create a local input file containing the statements you specify but not to pass the filename as part of the generated command line.
- filename* is the name of the local file. If you do not specify a name, **smake** uses the temporary filename **temp_smake.tmp**. If you specify a filename, enclose the filename in parentheses, and do not enter spaces before or after the filename.
- statement1-n* are the statements that you want included in the local input file.

The following example shows a smakefile that uses macros and a local input file to create the target **grep**. **grep** is a utility that is composed of C source files and one assembly language module, the startup module **c.a**.

```

1  # Generate an updated version of GREP
2
3  # Define macros
4  SCHDRS = grep.h pat.h
5  OBJ = obj/
6  FLAGS = IDIR=INCLUDE: IDIR=INCLUDE:sc/ OUT=$(OBJ)
7  SC = SC:sc
8
9  # Specify file dependencies for GREP executable, generate link
10 # command to be passed to AmigaDOS, then specify contents of
11 # WITH file.
12 grep: $(OBJ)grep.o $(OBJ)c.o $(OBJ)_main.o $(OBJ)re_gen.o
13 grep: $(OBJ)re_match.o $(OBJ)os.o
14     slink <WITH <
15 FROM $(OBJ)c.o+$(OBJ)grep.o+$(OBJ)_main.o+$(OBJ)re_gen.o
16 FROM $(OBJ)re_match.o $(OBJ)os.o
17 TO grep
18 LIBRARY LIB:sc.lib+LIB:amiga.lib

```

smake Maintains and updates records of file dependencies
(continued)

```

19  <
20
21  # Specify remaining file dependencies for file needed to generate
22  # GREP executable.
23  $(OBJ)grep.o: grep.c $(LCHDRS)
24      $(SC) $(FLAGS) define GREP grep
25
26  $(OBJ)c.o: c.a
27      asm -iinclude: -o$(OBJ)c.o c.a
28
29  $(OBJ)main.o: main.c
30      $(SC) $(FLAGS) define GREP main
31
32  $(OBJ)re_gen.o: re_gen.c $(LCHDRS)
33      $(SC) $(FLAGS) define GREP re_gen

```

Note: Do not enter line numbers in your smakefiles. Line numbers are included here for use in the discussion that follows.

Lines three through seven define macros. Lines 12 and 13 list the files that must be current before **smake** can remake the **grep** target. If a target has too many dependent files to fit on one line, retype the target filename and colon on the next line, and enter the remaining dependent filenames.

Line 14 tells **smake** to

1. create a local input file, using the default temporary filename, that contains all the lines up to the less than (<) sign in line 19
2. execute **slink** to generate the target **grep**
3. pass the local input file to **slink** as a local input file.

smake generates the following command and passes it to AmigaDOS:

```
slink WITH temp_smake.tmp
```

The temporary file **temp_smake.tmp** contains the following lines:

```

FROM $(OBJ)c.o $(OBJ)grep.o $(OBJ)_main.o $(OBJ)re_gen.o
FROM $(OBJ)re_match.o $(OBJ)os.o
TO grep
LIBRARY LIB:sc.lib+LIB:amiga.lib

```

smake Maintains and updates records of file dependencies
(continued)

Using Default (.def) Files Default files are smakefiles that **smake** searches whenever your smakefile does not specify what action is required to create a target. **smake** provides a default file called **smake.def**. To determine what action to take to create a target, **smake** looks for instructions in the following order:

1. actions specified below the dependency relation.
2. transformation rules in your smakefile.
3. **.DEFAULT** rules specified in your smakefile or in the **smake.def** file.
4. rules taken from an alternate **.def** file.
5. rules taken from a file named **smake.def** in the current directory.
6. rules internal to this implementation of **smake**. **smake** internal rules are the same rules that are in the default **smake.def** file.

However, editing **smake.def** does not change the internal rules.

Using the smake.def File

smake provides a default file, **smake.def**, that contains the basic command for compiling and linking a file, and it contains transformation rules for converting assembly language, C source code, and header files into object files. You can change these rules as necessary by editing **smake.def**.

smake.def contains the following:

```
SC = SC:sc
SCFLAGS = IDIR=INCLUDE:
ASM = SC:asm
LINK = SLINK:slink

.DEFAULT:
    $(SC) $(SCFLAGS) $*

.a.o:
    $(ASM) $* -iinclude: -o$*.o

.c.o:
    $(SC) $(SCFLAGS) $*
```

smake Maintains and updates records of file dependencies
(continued)

```
.h.o:
    delete *.o
    $(SC) $(SCFLAGS) $*
```

Creating and Using Your Own Default File

You can create your own default files. To use a `.def` file other than `smake.def`, specify the `-b` option on the **smake** command line as follows:

```
smake -bfilename.def
```

Your default file must have the file extension `.def`.

Running smake You can run **smake** from the Workbench or from the CLI.

To run **smake** from the Workbench, double-click on the **Build** icon. **smake** looks for the smakefile in your current directory. The default file names that **smake** looks for, in the order in which it looks for them, are as follows:

1. `smakefile`
2. `smakefile.smk`
3. `lmkfile`
4. `lmkfile.lmk`
5. `makefile`

To use a different smakefile name, specify the `-f` option as described later in this section. To specify options, define macros, or specify alternate targets for the **Build** icon, select the **Build** icon, choose **Information** from the Icons menu, and add the necessary parameters in the Tool Types box.

When running **smake** from the Workbench, the search order of files is based on the assignment of the device `smake_files:`. You can assign a value to this device using the following command (from the CLI):

```
assign smake_files: your-device:
```

When no assignment has been made, **smake** searches the current directory for the required files.

smake Maintains and updates records of file dependencies
(continued)

To run **smake** from the CLI, enter the **smake** command as follows:

```
smake [options] [macro definitions] [targets]
```

As with the **Build** icon, **smake** looks for one of the three default smakefile names in your current directory. To use a different smakefile name, specify the **-f** option as described later in this section.

smake supports the following options:

- a** rebuilds all targets and subtargets without regard to time stamps.
- bfile** uses the **.def** file you specify instead of the default file **smake.def**.
- c** tells **smake** to record everything it would have done if it had executed the command you entered. When you specify the **-c** option, **smake** does not execute the command you enter. Instead, it creates a batch file containing all the instructions it would have executed if you had not specified the **-c** option. This file is named **smakefile.bat** in the current directory, and you can run it by entering the following on the command line:


```
execute smakefile.bat
```
- d** prints detailed debugging information about the processing of the smakefile.
- e** erases any out-of-date targets before remaking them.
- f file** uses the filename you specify as the input smakefile. **smake** attempts to locate the file with the exact name you have specified. If this search is unsuccessful, **smake** searches for a file with the name you specified plus an extension of **.smake**. For example, suppose you have an executable named **alpha** and a smakefile named **alpha.smake** in the same directory. If you specify **smake -f alpha**, **smake** will assume that **alpha** is your smakefile. If you receive an error message such as **Line too long** or **Unexpected punctuation**, try renaming your file with an explicit **.smake** extension and running **smake** with the full filename.

smake Maintains and updates records of file dependencies
(continued)

- h prints help information and then exits.
- i ignores errors caused by executing the actions. Both the -k and -i actions should be used with caution because **smake** will continue running. The -k option incorporates all of the functionality of the -i option.
- k ignores error returns from actions passed to AmigaDOS and from **smake** not knowing how to make certain targets. Both the -k and -i actions should be used with caution because **smake** will continue running. The -k option incorporates all of the functionality of the -i option.
- n displays the actions **smake** would have taken, but **smake** does not execute these actions.
- p prints target descriptions and expanded macros.
- q determines if the target file is current. **smake** prints a 1 if the target is current or a 0 if it is not. **smake** will not execute any actions.
- s does not echo actions to the screen before executing them.
- t touches the target files by updating them with the current system time. **smake** does not execute any of the actions associated with these targets.
- u rebuilds unconditionally all targets and subtargets without regard to time stamps.
- x is for UNIX compatibility. If you specify this option, **smake** attempts to detect any features of the smakefile that would prevent it from working correctly with the UNIX **make** utility.
Examples of such features are:
 - using a local input file
 - defining an action line without an initial tab character
 - specifying multiple target files on a single target-dependency line.

In each case, **smake** displays a message describing the incompatibility.

splat Searches for and replaces patterns that match regular expressions

Synopsis `splat [>destination] [options] pattern string file . . .`

Description `splat` replaces all occurrences of *pattern* with *string* in each *file* that you specify. `splat` does not overwrite the original version of the file unless you specify the `-o` option. If you do not redirect the output by specifying a greater than (`>`) sign followed by a *destination*, `splat` places the changed file in either a temporary file or, if you use the `-d` option, in the specified directory. `splat` uses the same root filename with an extension of `.$$$`.

You must use the rules recognized by `grep` for specifying the *pattern* and *string*. For example, to specify a *string* that contains spaces, you must enclose the string in double quotes. For additional information, see the description of the `grep` utility earlier in this chapter.

`splat` supports the following options:

-d*directory*

specifies the directory where you want `splat` to place the new files. Using this option leaves the original versions untouched. If the specified directory does not exist, `splat` attempts to create it. If the attempt fails, `splat` displays an error message. The `-o` and `-d` options are mutually exclusive.

-o

tells `splat` to overwrite the original version of the file with the new version. The `-o` and `-d` options are mutually exclusive.

-s

displays the name of the file on which `splat` is currently working. It also displays a message saying that no substitutions have been performed if it was unable to find the specified pattern in the file. If you do not specify `-s`, `splat` performs its task silently.

-v

displays all lines in which substitutions are made, but it does not create new versions of the files. You can use this option to determine the changes that `splat` will make to your files.

Examples For example, suppose you want to substitute `INT` for `int` in every declaration of an integer identifier in your C source files. The following command replaces `int` with `INT` in all files with the `.headers` and `.source` extensions in your current directory. It writes the new versions of these files in the directory named `/backup`.

```
splat -d/backup/ int INT #?.headers #?.source
```


splat Searches for and replaces patterns that match regular expressions
(continued)

However, you do not want calls to `printf` to become `PRINTF`. Because in these declarations, the string `int` is followed by a space character, you should use the following command instead:

```
splat -d/backup/ "int " "INT " #?.headers #?.source
```

Similarly, the following command performs the same substitutions, but it writes the new versions of the files in the subdirectory `backup` of the current directory:

```
splat -dbackup/ "int " "INT " #?.headers #?.source
```

To avoid errors such as replacing `printf` with `PRINTF`, you can use the `-v` option to determine, without changing your source files or creating new files, if you have entered the correct command. The following command performs the same substitutions as the previous command and displays those changes on the screen, but it does not change the files:

```
splat -v "int " "INT " #?.headers #?.source
```

The following command replaces all occurrences of the string `unsigned char` with `char` within all files with the `.c` extension in the current directory:

```
splat "unsigned char" char #?.c
```

The following command appends the string `-cc` to the end of each line that contains the string `sc1` in the file `comp.bat`:

```
splat -dbak/ "sc1.*$" "% -cc" comp.bat
```

The pattern `sc1.*$` matches any string beginning with `sc1` followed by 0 or more of any character and ending with the end of the line. The string `% -cc` appends a space followed by `-cc` to the end of the line.

splat Searches for and replaces patterns that match regular expressions
(continued)

You can also use **splat** to insert lines in a file or to break single lines into multiple lines. For example, you can have a file named **text.c** that contains the following line:

```
parms(s);usage();exit(1);
```

Suppose you want the calls to **usage** and **exit** to be on separate lines and indented one tab stop; enter the following command (all on one line):

```
splat parms(s);usage();exit(1); parms(s);\n\tusage();  
\n\texit(1);\n\ttext.c
```

tb Displays traceback information

Synopsis **tb** [>destination] [options] [tbfile [program]]

Description The **tb** utility processes the traceback file that is created when you link your program with **catch.o** and your program terminates abnormally.

To get traceback information for your program, you must link your program with **catch.o**. If your program terminates abnormally, **catch.o** creates a standard IFF format file named **Snapshot.TB** in your current directory. This file contains up to six sections:

- ☐ symbol cross reference
- ☐ stack
- ☐ registers
- ☐ environment
- ☐ memory
- ☐ user data.

Each section contains information about your program at the time it terminated. (Two sections, memory and user data, may not be available for your program.)

tb displays the traceback information on the screen unless you redirect it by specifying a greater than (>) sign followed by a destination.

If you do not specify any options, **tb** displays only the current stack frame, which indicates the location where the program terminated. By specifying options, you can print all of the sections in the traceback file.

By default, **tb** looks for the traceback file **Snapshot.TB** in your current directory. If you want **tb** to use a different traceback file, specify the traceback file as the *tbfile* parameter.

The *program* parameter specifies the program from which you want **tb** to read debugging information. If you do not specify *program*, this utility uses the program specified in the traceback file.

Note: Specifying a *program* is useful when you have two executables of your program: one created with debugging information and another without debugging information. The version without debugging information loads faster, but you still have access to the traceback information if your program crashes.

tb Displays traceback information
(continued)

tb supports the following options:

- l displays all sections present in the traceback file
- x displays the location of all symbols encountered in the program
- s displays the contents of the entire stack at the time your program terminated
- r displays the current contents of registers at the time your program terminated
- v displays the callback chain and the location where the program terminated
- m displays the amount of memory available at the time your program terminated (if present in **Snapshot.TB**)
- u displays any user data generated by your program (if present in **Snapshot.TB**).

Examples Suppose you have a program named **testit** that you link with **catch.o**, and **testit** terminates abnormally when you run it. You can display information about where the program terminates by entering the **tb** command without any options, as follows:

```
1> tb
Program Name: testit; run from CLI
Program load map (addresses are APTRs, sizes are in bytes)
220f78 $1110 211268 $5b4
Terminated with GURU number 00000005, Divide by Zero Error
Error occurred at address 221cde = _foo line 5
    called from 22156a = main line 11
    called from 221c60 = _main + 704
    called from 2210ca = hunk 0 + $152
```

However, the file created by **catch.o** contains much more information. To display all of the information contained in **Snapshot.TB**, you can specify the **-l** option, as follows:

```
1> tb -l
TraceDump 0.88 Copyright (c) 1988 The Software Distillery
```

tb Displays traceback information
(continued)

```
TraceDump Utility: catch.o; Version 1, Revision 0
Processor type: 68000
VBlankFreq 60, PowerSupFreq 60
```

```
Symbols for hunk 0
      _foo = 22153c                _main = 22155c
```

```
Program Name: testit; run from CLI
Program load map (addresses are APTRs, sizes are in bytes)
220f78 $1110 211268 $5b4
Terminated with GURU number 00000005, Divide by Zero Error
Error occurred at address 221cde = _foo line 5
    called from 22156a = main line 11
    called from 221c60 = _main + 704
    called from 2210ca = hunk 0 + $152
```

```
Registers:
D0=00221cde D1=00000000 D2=00000001 D3=00002718
D4=00000001 D5=0000002b D6=0000003b D7=00000000
A0=00221f54 A1=00243fcc A2=0024460a A3=00225c68
A4=00211268 A5=002445be A6=002033c8 A7=002445ae
PC 221cde  C=0 V=0 Z=1 N=0 X=0
```

```
stack top: 244640, stack pointer: 2445ae, stack length: $94
entire stack, size = $94 bytes
2445AE: 000003ED 00221558 0000002B 00000000 : ...m.".X...+....
2445BE: 002445CA 0022156A 00225C68 002445F6 : .$EJ.".j."h.$E.
2445CE: 00221C60 00000001 00211620 00225CC4 : .".'.....!. ."D
2445DE: 00226030 00225C68 83D10000 00000021 : ." '0."h.Q.....!
2445EE: 13230021 00211620 00244640 002210CA : .#.!.!. .$F0."J
2445FE: 00244602 74657374 6974000A 00000022 : .$F.testit....."
24460E: 60300000 27100000 27180000 00010000 : '0..'...'.....
24461E: 002B0000 003B0022 60300022 61300020 : .+...;.'"0."a0.
24462E: 35300022 0F740024 464400FF 425800FF : 50."t.$FD..BX..
24463E: 424C0022                                : BL."
```

■ Part 3

Using the SAS/C[®]

Macro Assembler

Chapter 11 Using Assembly Language with C Language

11 Using Assembly Language with C Language

299	<i>Introduction</i>
300	<i>Writing Assembly Language Functions</i>
300	<i>Writing Assembler Statements</i>
304	<i>Using Assembler Directives</i>
308	<i>Defining and Using Macros</i>
309	<i>Defining Control Sections</i>
314	<i>Communicating between Assembly Language and C Language</i>
314	<i>Calling Assembler Functions from C Functions</i>
322	<i>Calling C Functions from Assembler Functions</i>
322	<i>Referencing Global Data</i>
323	<i>Running the Assembler</i>

Introduction

The SAS/C Macro Assembler reads an assembly language source file and produces an object file in the Amiga object file format. You can use the assembler to generate assembly language modules for your C programs, or you can generate programs written entirely in assembly language. For example, some operations such as directly accessing CPU and I/O registers cannot be handled easily in the C language. Also, assembly language is sometimes necessary to get the best combination of code size and speed.

The assembler supports the full set of 68xxx mnemonics, an extensive set of assembler directives, and a powerful macro facility. To use the assembler, you should understand thoroughly the Motorola 68xxx instruction set. The “Additional Documentation” section in “Using This Book” lists books that describe the Motorola 68xxx instruction set. If you are writing assembly language modules to be called from C functions, or if your assembly language modules call C functions, you should also understand the SAS/C function protocol. This chapter assumes that you have a knowledge of assembler, but not necessarily the SAS/C Macro Assembler.

This chapter describes the directives supported by the assembler, discusses how to write and use macros, and describes how to call C functions and assembler modules.

Writing Assembly Language Functions

The following sections provide an overview of the syntax expected by the SAS/C Assembler. The topics covered include

- the format of assembler statements
- the directives supported by the Assembler
- how to create and use macros
- how to define control sections
- how to specify addresses in assembler statements
- how to specify floating-point data
- the offsets supported by branch instructions.

Writing Assembler Statements

Each assembly language source line has the following format:

[label:] operation[.size] operand . . . [;comment]

You can enter white space (spaces or tabs) before any field. You must enter white space between the operation and operand. The fields of the source line are described below:

label

can start in any column and end with a colon, or can start in column one and end with either a colon or white space. A label can be up to 31 characters long and can contain letters, digits, underscores (_), and dollar (\$) signs. A label cannot start with a digit, and the case of letters is significant. For example, the assembler differentiates between **XYZ**, **xYZ**, and **xyz**.

operation

is the name of an instruction, a directive, or a macro. Do not begin an operation in column one. If no label is present, precede the operation with white space. If a label is present, you must include a colon or white space between the label and operation. The case of this field is not significant. For example, **MOVE** is the same as **move**.

size

specifies either the size of the operands or the offset, depending on the type of instruction. See the following section, “Specifying Sizes,” for additional information.

operand

may contain 0 or more expressions, depending on the specific operation. Expressions are composed of constants, variables, and operators. The information in the following section, “Specifying Expressions,” describes constants and operators in more detail.

comment

can be entered in your assembler source code by preceding the comment text with a semicolon.

Specifying Sizes

Support for size suffixes depends on the instruction that you enter and the processor that you use. Check the documentation for your processor to determine which size suffixes are valid. For most non-floating-point instructions, the following suffixes are valid:

Suffix	Meaning
<i>none</i>	defaults to word (two bytes)
B	byte
L	long
S	short (same as byte)
W	word

For floating-point instructions, the following suffixes are valid:

Suffix	Meaning
<i>none</i>	defaults to word
B	byte
D	double
L	long
P	packed
S	single
W	word
X	extended

For branch instructions **BSR** and **BRA**, the sizes supported by the assembler are as follows:

Suffix	Meaning
<i>none</i>	calculates the size offset needed. For externs, the assembler generates a 16-bit offset.
S	generates 8-bit offsets.
B	generates 8-bit offsets.
W	generates 16-bit offsets.
L	generates 32-bit offsets. BSR.L is supported only on the 68020 and higher processors, and you cannot use BSR.L to branch to externs.

For example, the following instruction creates a 32-bit branch:

```
BRA.L subpr
```

Data are converted into single, double, or extended precision according to the type of instruction. For example, the following instruction converts 2.1 into single precision and stores it in floating-point register 1:

```
fmove.s #2.1,fp1
```

As an additional example, the **fmove** instructions for the decimal number 2.1 specifying the bit pattern in hexadecimal are as follows:

```
fmove.s #$40066666,fp1
fmove.d #$4000CCCCCCCCD,fp1
fmove.x #$400000008666666666666666,fp1
```

Specifying Expressions

As described earlier, an expression is composed of constants, variables, and operators.

A *variable* is a label name or a name defined using an assembler directive.

A *constant* is a decimal, hexadecimal, octal, or binary number, or an ASCII literal. Unless you specify otherwise, the assembler expects a decimal number. To specify an ASCII literal, enclose the literal in single quotes. To specify a hexadecimal number, precede the number with a dollar sign. To specify an octal number, begin the number with an at (@) sign. To specify a binary number, begin the number with a percent (%) sign. The following table shows how to enter different types of constants:

Type or Base	Prefix	Example
decimal	<i>none</i>	1234
ASCII literal	' <i>literal</i> '	'AC9T'
hexadecimal	\$	\$89AB
octal	@	@743
binary	%	%1011
floating point	<i>none</i>	2.1
scientific notation	<i>none</i>	2.1E+10
hexadecimal floating point	\$	\$4066666

Note: If you use hexadecimal notation to specify floating-point numbers, you must use either the IEEE or FFP bit pattern representation. For information about this representation, refer to a manual for the 6888x math coprocessor.

The following table lists the *operators* recognized by the assembler in the order in which they are processed.

Order	Operator	Meaning
1	—	unary minus
2	>>	right shift
	<<	left shift

(continued)

Order	Operator	Meaning
3	*	multiply
—	/	divide
—	%	modulus
4	+	add
—	—	subtract
5	<	less than
—	<=	less than or equal to
—	>	greater than
—	>=	greater than or equal to
—	==	equal to
—	!=	not equal to
6	&	bitwise AND
—	!	bitwise OR

For example, in the following expression, **PDQ** is first negated and then multiplied by **DEF**, and the result is added to **ABC**:

ABC+DEF*-PDQ

Each expression represents a 32-bit value. An *absolute expression* is one that contains only constants. A *relocatable expression* contains symbols whose values are determined during the linking and loading procedure.

**Using
Assembler
Directives**

With the exception of the **DC**, **DCB**, and **DS** directives, an assembler *directive* is an instruction to the assembler instead of an instruction to be translated into object code. Directives cannot begin in the first character of the input line.

The assembler supports the following directives:

[label] CNOP offset,alignment

aligns any data structure or entry point to any boundary. *alignment* is the alignment required for the base, such as 4 for long word, and *offset* is the offset from that alignment.

CSECT *name,type,align,rtype,rsiz*

defines section type and alignment requirements. The section “Defining Control Sections,” later in this chapter, discusses the **CSECT** macro.

[label] **DC***[.size] expression[,expression] . . .*

defines a constant value in memory.

[label] **DCB***[.size] n,expression*

defines a constant block in memory. Using the **DCB** directive is equivalent to using the **DC** directive *n* times.

[label] **DS***[.size] n*

reserves (defines) *n* memory locations of the size indicated by *.size*.

ELSE

begins an alternative for the **IF** directive.

[label] **END**

defines the end of the program.

ENDC

defines the end of conditional assembly. This directive turns off each of the **IFxx** directives.

ENDM

ends a macro definition.

label **EQU** *expression*

assigns permanently the value of *expression* to *label*.

label **EQU** *register*

equates an address or data *register* with *label*.

EXITM

exits the macro expansion. This directive is a synonym for **MEXIT**.

FAIL

generates an error for the next input line.

FORMAT

is included for compatibility only.

IDNT *name*

defines a program unit name.

IF *expression*

assembles the following statements to the next **ELSE** or **ENDIF** if the expression supplied as its argument is not 0. **IF** statements may be nested.

IFC *string,string*

assembles if the strings are identical.

IFD *symbol*

assembles if the symbol is defined.

IFEQ *expression*

assembles if the expression is 0.

IFGE *expression*

assembles if the expression is greater than or equal to 0.

IFGT *expression*

assembles if the expression is greater than 0.

IFLE *expression*

assembles if the expression is less than or equal to 0.

IFLT *expression*

assembles if the expression is less than 0.

IFNC *string, string*

assembles if the strings are not identical.

IFND *symbol*

assembles if the symbol is not defined.

IFNE *expression*

assembles if the expression is not equal to 0.

INCLUDE *"filename"*

includes external files into the source. You can nest **INCLUDE** directives up to a depth of 3.

LIST

tells the assembler to produce a listing file.

LLEN *n*

sets the number of characters per line in the listing file. *n* can be any value from 60 to 132. The default value is 132 characters.

label **MACRO**

begins a macro definition. The following section, “Defining and Using Macros,” describes how to define and use macros.

MASK2

is included for compatibility only.

MEXIT

exits the macro expansion. This directive is a synonym for **EXITM**.

NARG

is the reserved symbol to which the assembler assigns the index of the last argument passed to the macro in the parameter list.

NOFORMAT

is included for compatibility only.

NOLIST

tells the assembler to stop producing a listing file.

NOPAGE

tells the assembler to stop inserting page breaks and title headers in the listing file.

NOOBJ

stops the production of the object file.

OFFSET *n*

defines offsets. To define a table of offsets using the **DS** directive, you use the **OFFSET** directive. To end an offset section, use a **RORG**, **OFFSET**, **SECTION**, or **END** directive.

OPSYN *oldname newname*

defines a synonym for opcodes.

PAGE

inserts a page break into the listing file.

PLEN *n*

sets the number of lines per page in the listing file. *n* can be any value from 24 to 100. The default is 60 lines.

***label* REG *R1*[-*R2*][/*R3* [-*R4*]] . . .**

assigns permanently the register list to *label*. You can specify individual registers by separating each register with a slash (*R1/R2/R3*), or you can specify a range of registers by specifying the first and last registers separated with a hyphen (*R1-R3*). You can also combine individual registers with a range of registers (*R1/R3-R5/R7*). The assembler translates this list into the register list mask format used by the **MOVEM** instruction.

[*label*] RORG *n*

sets the program counter to *n* bytes from the start of the current relocatable section.

[*label*] SECTION *name*[,*type*]

defines a program section. *name* is a character string, and *type* must be **CODE**, **DATA**, or **BSS**. The section “Defining Control Sections,” later in this chapter, discusses the **SECTION** macro.

label* SET *expression

assigns temporarily the value of the *expression* to the symbol identified by *label*.

SPC *n*

skips *n* blank lines in the listing file.

TTL *title*

sets the title string that is printed in the page headers in the listing file. The title can be no longer than 40 characters.

XDEF *label[,label]* . . .

defines an internal label as an external entry. By using the **XREF** directive, you can then refer to the symbols generated by these labels from other modules and from routines written in the C language.

XREF *label[,label]* . . .

references an external symbol defined with the **XDEF** directive.

Defining and Using Macros

You can define a macro using the following sequence of directives and instructions:

```
label MACRO
.
.
.
ENDM
```

You must begin the definition with a **MACRO** directive on a line by itself. After the **MACRO** directive, enter the lines that constitute the macro procedure. End the macro definition with the **ENDM** directive on a line by itself. You can use the **EXITM** directive within the macro procedure to exit the macro before the **ENDM** directive.

For example, you could define a macro called **acopy** as follows:

```
acopy MACRO
    move.w    4(sp),\1
    move.l    6(sp),\2
ENDM
```

You can invoke this macro with the following statement:

```
acopy    d0,a5
```

The assembler expands the macro as follows:

```
move.w    4(sp),d0
move.l    6(sp),a5
```

Defining Control Sections

The SAS/C Macro Assembler does not assume any default control sections. You must use either the **SECTION** or **CSECT** directive before you define any code or data. A **SECTION** or **CSECT** directive defines the beginning of a relocatable control section. Parameters on these directives allow you to name each control section. The assembler uses these named control sections to control the location of data and code during linking.

Chapter 12, “How Does the Compiler Work?,” in *SAS/C Development User’s Guide, Volume I*, describes the sections produced by the compiler.

Using the SECTION Directive

The **SECTION** directive follows this format:

```
[label] SECTION name[, type]
```

The following list describes each of the parameters.

- name* specifies the name of the control section. You can use the name **__MERGED** for near data. You can use a specific name only once in a given **asm** file.
- type* specifies the type of section. The type can be **BSS**, **DATA**, or **CODE**.

Specifying Addresses with the SECTION Directive

You can define symbols to be in the data section by using **XREF** and **XDEF** after the **SECTION** directive. The following example uses 16-bit addressing to reference both code and data:

```

SECTION    __MERGED, DATA
XREF      xdata
data1     DC.W      10
data2     DC.L      20
.
.
.
SECTION    TEXT, CODE
XREF      cfunc
XDEF      afunc
afunc     JSR        cfunc(PC)
          MOVE.W     data1(A4), D0
          MOVE.L     data2(A4), D1
          MOVE.L     d1, xdata(A4)
```

To use 32-bit addressing, remove references to the register A4 and to the Program Counter (PC).

Using the **CSECT** Directive

The **CSECT** directive follows this format:

```
CSECT name[, type][, align][, rtype][, rsize]
```

The **name** and **type** parameters describe the control section being defined. The **rtype** and **rsize** parameters define how to access the information in the section. The **align** parameter is ignored. The following list describes each of the parameters:

name specifies the name of the control section. You can use the name **__MERGED** for near data. You can use a specific name only once in a given **asm** file.

type specifies the type of section. A value of 0 indicates a code section, 1 indicates a data section, and 2 indicates an uninitialized data section (like the **udata** section used by the compiler). The default value is 0 (code).

align specifies the alignment requirements of the control section as a power of 2. For example, an alignment of 1 means that the section will begin on an even address, and a value of 2 means that the section will begin at an address divisible by 4.

Note: This parameter is not used on the Amiga. All sections are long-word aligned.

rtype specifies whether the code or data in the section are addressed relative to register A4 or to the PC. The default value is 0.

rsize specifies the size of the offset for the address. If you specify an *rsize* of 4, the assembler uses absolute addressing. If you specify an *rsize* of 2, the assembler uses the value of *rtype* to determine how to address code or data in the section. The default value is 4.

The following table summarizes the valid types and sizes of relocation information on the 68000:

<i>rtype</i>	<i>rsiz</i> e	Description
0, 1, 2	4	32-bit absolute address
0	2	PC-relative address (code section)
1	2	A4-relative address (data section)
2	2	A4-relative address (BSS section)

Specifying Addresses with the CSECT Directive

By specifying parameters in the **CSECT** directive, you can generate assembler modules that are compatible with C programs compiled with the **data=near** (base-relative data) and **code=near** (program counter-relative addressing) compiler options.

You can force address register-relative addressing of data in the BSS section by specifying an **rtype** of 2 and an **rsiz**e of 2. Any **XREF** statements for external data elements must appear after the **CSECT** statement, and all data must be addressed relative to register A4. For example, the following code moves data around:

```

                                CSECT  __MERGED,1,,2,2
                                XREF    xdata
data1    DC.W    10
data2    DC.L    20
.
.
.
                                CSECT  TEXT,0
                                MOVE.W  data1(A4),D0
                                MOVE.L  data2(A4),D1
                                MOVE.L  d1,xdata(A4)

```

Note: If the **XREF** statement for the external data precedes the **CSECT** statement, you can use a standard reference to a four-byte external address.

You can force program counter-relative addressing of data in the code section by specifying an **rtype** of 0 and an **rsiz**e of 2. The **XREF** statement for the external function must appear after the **CSECT**

statement, and the function must be called relative to the program counter. For example, the following code calls the function `cfunc` relative to the PC:

```

        CSECT  TEXT,0,,0,2
        XREF   cfunc
        XDEF   afunc
afunc    JSR    cfunc(PC)
        RTS
```

Note: If the `XREF` statement for the external function precedes the `CSECT` statement, you can use a standard `JSR` instruction to a 4-byte external address.

In addition to specifying addresses relative to address registers and to the program counter, you also can specify absolute addresses or enter immediate data. The following tables summarize the addressing modes supported by the SAS/C Assembler and provide examples of each mode. The notation used in these tables is as follows:

Notation	Meaning
<i>An</i>	an address register (a0—a7)
<i>bd</i>	a 32-bit byte displacement
<i>data</i>	actual data you want loaded into the register
<i>Dn</i>	a data register (d0—d7)
<i>d⁸</i>	an 8-bit displacement
<i>d¹⁶</i>	a 16-bit displacement
<i>od</i>	the 32-bit outer displacement
<i>PC</i>	program counter
<i>Xn</i>	an index register (a0.w—a7.l)
<i>xxx</i>	any number in decimal, hexadecimal, octal, or binary format

You can specify the displacements, data, and numbers in decimal, hexadecimal, octal, or binary format. The information on entering operands in the section “Writing Assembler Statements,” earlier in this chapter, describes how to specify each type of constant.

Note: All fields of the operands on the addressing modes marked 68020 are optional.

The following table shows examples of register-direct addressing modes:

Mode	Example
<i>Dn</i>	<code>add.w d1,d0</code>
<i>An</i>	<code>addq.w #1,a1</code>

The following table shows examples of register-indirect addressing modes:

Mode	Example
<i>(An)</i>	<code>add.w (a1),d0</code>
<i>(An)+</i>	<code>add.w (a1)+,d0</code>
<i>-(An)</i>	<code>add.w -(a1),d0</code>
<i>d¹⁶(An)</i>	<code>add.w 10(a1),d0</code>
<i>d⁸(An,Xn)</i>	<code>add.w 10(a1,a2.1),d0</code>
<i>bd(An,Xn)</i>	<code>add.w \$10000(a1,a2.1),d0</code> (68020 only)
<i>([bd,An],Xn,od)</i>	<code>add.w ([10,a1],a2.1,20),d0</code> (68020 only)
<i>([bd,An,Xn],od)</i>	<code>add.w ([10,a1,a2.1],20),d0</code> (68020 only)

The following table shows examples of absolute addressing modes:

Mode	Example
<i>(xxx).W</i>	<code>add.w (100).w,d0</code>
<i>(xxx).L</i>	<code>add.l (100).l,d0</code>

The following table shows how to specify immediate data:

Mode	Example
<code>#data</code>	<code>add.l #100,d0</code>

The following table shows examples of program counter-relative addressing modes:

Mode	Example
<code>d¹⁶(PC)</code>	<code>add.w 10(PC),d0</code>
<code>d⁸(PC,Xn)</code>	<code>add.w 10(PC,a2.1),d0</code>
<code>bd(PC,Xn)</code>	<code>add.w \$10000(PC,a2.1)</code> (68020 only)
<code>([bd,PC],Xn,od)</code>	<code>add.w ([10,PC],a2.1,20),d0</code> (68020 only)
<code>([bd,PC,Xn],od)</code>	<code>add.w ([10,PC,a2.1],20),d0</code> (68020 only)

Communicating between Assembly Language and C Language

The following sections describe how to call assembler functions from C functions and how to call C functions from assembler modules.

Calling Assembler Functions from C Functions

Assembler modules may be called from C functions in three ways:

- passing parameters by placing them on the stack. Unless you specify the `parameters=register` option or use the `__asm` keyword, C functions pass parameters on the stack.
- compiling your program with the `parameters=register` option. This option tells the compiler to pass parameters in registers, but it allows the compiler to determine which registers to use.
- using the `__asm` keyword and specifying which registers parameters are placed in.

Passing parameters in registers produces a faster, smaller executable module.

Passing Parameters on the Stack

As previously stated, C functions pass parameters on the stack if you do not specify the `parameters=register` option or use the `__asm` keyword.

For example, the following assembler module, `func`, adds or subtracts an integer argument and a double argument depending on whether a plus (+) sign or a minus (-) sign is passed in the character argument.

```
; NOTE: This code assumes that you have a math co-processor.
;       If you do not, your machine will crash when you run
;       this program.

xdef    _func

section code

_func:
    fmove.d 8(a7),fp0           ; middle

    cmpi.b  #'+',7(a7)          ; should I add?
    bne.b   noadd               ; no, check subtract

    fadd.l  16(a7),fp0          ; add 'mrright' to 'middle'
    bra.b   exitpoint          ; ready to return

noadd:
    cmpi.b  #'-',7(a7)          ; should I subtract?
    bne.b   nosub               ; no, exit with error

    fsub.l  16(a7),fp0          ; subtract 'mrright'
    bra.b   exitpoint          ; from 'middle'

nosub:
    fmove.b #-1,fp0             ; return -1 for error

exitpoint:
    fmove.l fp0,d0              ; return 'int' in d0
    rts

end
```

The `func` assembler module returns an integer.

To call an assembler module from your C program, include the prototype for the assembler module in your C source file before calling the function. The prototype for the previous example is the following:

```
int func (char lefty, double middle, int mrright);
```

For example, you can write the following program to call the `func` module:

```
#include <stdio.h>

int func(char lefty, double middle, int mrright);

void main(void)
{
    char lefty;
    double middle;
    int mrright;

    lefty = '+';
    middle = 12345.;
    mrright = 99;

    printf("func(+) = %d\n", func(lefty, middle, mrright));
}
```

If you save the assembler module `func` in a file named `testa.a`, you can assemble `test.a` with the following command:

```
asm -m2 testa.a
```

If you save the C source code in a file named `test`, you can compile and link `test` with the following commands:

```
sc math=868881 test
slink lib:c.o test.o testa.o lib lib:sc.lib
```

The `test` program will pass parameters on the stack. When you run `test`, you should see the following output:

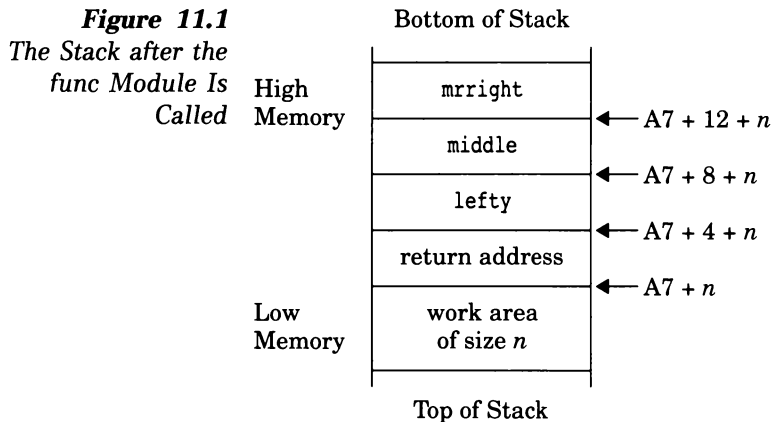
```
func(+) = 12444
```

To use the parameters inside your assembler module, you must retrieve the parameters from the stack. The arguments are placed on the stack from right to left; that is, the leftmost argument is immediately above the 4-byte return address in the stack. For example, in the following function call, the compiler generates code to push the arguments `mrright`, `middle`, and then `lefty` onto the stack.

```
char lefty;
double middle;
int mrright;

func(lefty,middle,mrright);
```

Next, the `func` module is called, and the return address is pushed onto the stack. Figure 11.1 shows the stack after the `func` module has been called.



The arguments `mrright`, `middle`, and `lefty` use 4 bytes, 8 bytes, and 4 bytes, respectively.* Register A7 is the stack pointer, and it points to the 4-byte return address.

Note: If you specify the `shortintegers` option, integers will be 2 bytes instead of 4.

Before retrieving arguments from the stack into non-scratch registers, you must first save the contents of those registers, so they

* The smallest item that can be pushed onto the stack is the size of an integer. The size depends on whether you specify the `shortint` compiler option. A `char` will be promoted to an `int`.

may be restored when your function terminates. Scratch registers are registers D0, D1, A0, A1, FP0, and FP1.

The following list describes the steps your assembler module should take to save register values and establish frame pointers. In these steps, `n` refers to the size of your work area in bytes.

- 1. Enter the SUBI instruction if you need local storage:

```
SUBI    n,A7
```

- 2. Save registers D2 through D7, A2 through A6 if they will be changed during execution of the module.
- 3. Save floating-point registers FP2 through FP7 if a 68881 math coprocessor is present and will be used by the module.

You can address the arguments from A7 as follows:

Location	Size	Contents
(A7)+n	4	return address
(A7)+4+n	4	argument <code>lefty</code>
(A7)+8+n	8	argument <code>middle</code>
(A7)+16+n	4	argument <code>mrright</code>

Using the `parameters=register` Option

Using the `parameters=register` option tells the compiler to pass some of the function arguments in registers instead of on the stack. Specifically, the first two pointer arguments are in A0 and A1, and the first two integral arguments are in D0 and D1. All other arguments are passed on the stack.

The compiler identifies functions that expect arguments in registers by placing an `@` in front of the function name. The `@` replaces the underscore (`_`) that the compiler normally places at the beginning of function names.

Using the **parameters=register** option is similar to passing parameters using the stack. For example, make sure you:

- include a prototype of your assembler module in your C source program.
- retrieve any parameters that are passed on the stack before using them.
- use the same **asm** command to assemble your assembler code.
- use the same **slink** command to link your C code.

However, you must specify the **parameters=register** option in the **sc** command as follows:

```
sc parameters=register test
```

Using the **__asm** Keyword

Using the **__asm** keyword on a function prototype or definition tells the compiler to pass parameters in specific registers.

When you use **__asm**, the calling C function places the parameters in the registers required by the assembler module, so no stack manipulation inside the assembler module is necessary.

The following assembler code is the same **func** module that is described previously in the section “Passing Parameters on the Stack,” but it is modified to use the **__asm** keyword.

```
; NOTE: This code assumes that you have a math co-processor.
;       If you do not, your machine will crash when you run
;       this program.

xdef    _func

section code

_func:
    cmpi.b    #'+',d0        ; should I add?
    bne.b     noadd          ; no, check subtract

    fadd.l    d1,fp0         ; add 'mrright' to 'middle'
    bra.b     exitpoint      ; ready to return

noadd:
    cmpi.b    #'-',d0        ; should I subtract?
    bne.b     nosub          ; no, exit with error
```

```

        fsub.l  d1,fp0          ; subtract 'mrright' from 'middle'
        bra.b   exitpoint

nosub:
        fmove.b #-1,fp0        ; return -1 for error

exitpoint:
        fmove.l fp0,d0         ; return 'int' in d0
        rts

        end

```

To use the `--asm` keyword, you must modify the prototype of the assembler module in your C source file. For each parameter that you want to pass in a register, precede the parameter with the `register` keyword, two underscores (`--`), and the register into which you want the parameter placed.

The prototype for the previous example is the following:

```

int __asm func(register __d0 char lefty,
               register __fp0 double middle,
               register __d1 int mrright);

```

The `lefty` argument is placed into register `d0`, the `middle` argument is placed into register `fp0`, and the `mrright` argument is placed into register `d1`.

The remainder of your C program is the same as that shown earlier in the section “Passing Parameters on the Stack.” You also use the same `asm`, `sc`, and `slink` commands as shown in the section “Passing Parameters on the Stack.”

Returning Values to the Calling Function

Your assembler module should return values in one or more registers. The register(s) depend on the data type of the calling function. Function return values are passed back in one or more registers,

depending on the data type declared for the calling function. The conventions are as follows:

Return Type	Length (Bits)	Syntax	Meaning
char	8	D0.B	low byte of D0
short	16	D0.W	low word of D0
long	32	D0.L	all of D0
pointer	32	D0.L	all of D0
float	32	D0.L	all of D0
float (68881)	32	FP0.S	single precision in FP0
double (IEEE)	64	D0.L,D1.L	high bits in D0
double (FFP)	32	D0.L	all of D0
double (68881)	96	FP0.X	all of FP0

If the assembler module returns a structure or union, it must define a static work area (not on the stack) to hold the returned object and return a pointer to this object in the D0 register. The calling function should immediately move the data to the appropriate place.

Note: This approach implies that functions returning structures or unions are not re-entrant, although they are serially reusable. Such a function can be recursive if it is designed carefully with this return technique in mind.

After setting up the return value, the assembler module exits by reversing the steps described earlier:

1. restore registers that were saved.
2. advance the stack pointer in register A7 past the work area.
3. return to the caller using an **RTS** instruction.

The calling function removes the arguments from the stack.

Calling C Functions from Assembler Functions

The steps you take to call a C function from an assembler program vary, depending on whether or not you compile your C function using the `parameters=register` option.

If you are not using registerized parameters, follow these steps:

1. push the arguments on the stack in the correct order.
2. call the function using the `JSR` instruction.
3. after control returns from the called function, adjust the stack pointer to account for pushed arguments.

For example, the following code calls the function `cfunc` with two parameters that are currently in registers D7 and D6 and stores the return code in the `ret` variable.

```
XREF      cfunc
MOVE.L    D7,-(A7)      ;push argument
MOVE.L    D6,-(A7)
JSR       cfunc         ;call function
ADDQ      #8,A7         ;restore stack ptr
MOVE.L    D0,ret        ;save return value
```

If you are using registerized parameters, follow these steps:

1. place the appropriate arguments in the correct registers.
2. push any remaining arguments onto the stack.
3. add an `@` to the front of the function name, and call the function using the `JSR` instruction.
4. after control returns from the called function, adjust the stack pointer to account for pushed arguments (if any).

For example, the following code calls the function `cfunc` with two parameters that are currently in registers D7 and D6 and stores the return code in the `ret` variable.

```
XREF      @cfunc
MOVE.L    D7,D1
MOVE.L    D6,D0
JSR       @cfunc        ;call function
MOVE.L    D0,ret        ;save return value
```

If your C function is declared using the `__asm` keyword, you should pass the parameters in the appropriate registers.

Referencing Global Data

Instead of passing parameters into and out of a function, you can use global data.

You can access data declared in an assembler module from a C function by placing the data declarations inside a data section and defining the data with an **XDEF** statement. The following code defines two variables in assembly language and shows how to declare them in a C function.

In Assembly Language	In C Language
CSECT data	extern long VAR1;
XDEF VAR1,VAR2,VAR3	extern long VAR2;
VAR1 DC.L \$4000	
VAR2 DC.L \$8000	

You can access data declared in a C function from an assembler module using the **XREF** statement, as follows:

In C Language	In Assembly Language
long VAR1=0x4000;	XREF VAR1,VAR2
long VAR1=0x4000;	.
	.
	.
	MOVE.L VAR1,D0

Running the Assembler

After you have created your modules or programs in assembly language, you can run the Assembler by entering the **asm** command as follows:

```
asm [>listfile] [options] filename
```

The *filename* is the name of the assembly language source file that you want to assemble. The assembler assumes a source filename extension of **.a**, and it produces an object file with the same filename but with the **.o** extension. You can then use the linker, **slink**, to

combine these object modules into an executable file, called a *load module*.

If you include *>listfile*, the assembler sends the source listing and error messages to that file.

The assembler supports the following options:

-c *x*

specifies that the sections designated by *x* are to be loaded into memory addressable by the Amiga system's custom hardware. You should specify **-c** for screen image and audio data. The *x* can be one or more of the following:

- b** bss or uninitialized data
- c** code segment
- d** data segment

By default, the assembler loads the segments into memory not addressable by custom hardware, if it is available. This action avoids bus contention between the processor and the custom hardware. For example, the following command causes all segments to be loaded into chip-addressable memory, regardless of the system memory configuration:

```
asm -ccdb prog.a
```

-d

activates the debugging mode.

-d *symbol* [*=value*]

defines *symbol* as if your source file contained this statement:

```
symbol EQU value
```

If you do not specify a *value*, the assembler assigns a value of 1.

-h *x*

specifies that the sections designated by *x* are to be loaded into memory not addressable by the Amiga system's custom hardware. The *x* can be one or more of the following:

- b** bss or uninitialized data
- c** code segment
- d** data segment

By default, the assembler loads the segments into memory not addressable by custom hardware, if it is available. This action

prevents the specified segments from being loaded into chip-addressable memory even if no other memory is available. This action could also prevent programs from running if external high-speed memory is not available on the machine. For example, the following command loads all three sections into high-speed RAM:

```
asm -hcdb prog.a
```

-iprefix

specifies a prefix that should be placed in front of filenames in **INCLUDE** directives. You can enter up to four **-i** options. The assembler will search for the files in the order in which you specify the prefixes. If you specify a directory name as a prefix, include the trailing slash (/). For example, you can assemble the file named **testprog.a** with the following command:

```
asm -imyinc/ -iyourinc/ testprog.a
```

If your program contains the statement **include abc.i**, the assembler searches for the file in the current directory. If that fails, it searches for **myinc/abc.i** and then **yourinc/abc.i**.

-l[x][m][i]

sends a listing of the source file to the standard output, usually the screen. The listing displays the appropriate location counter and code- or data-generated information beside the source line. The **x**, **m**, and **i** produce the following slightly different listings:

- x** lists the expansion text for macros.
- m** lists additional data generated for source lines that cannot be listed beside the original source line. This option allows multiple listing lines for each source line.
- i** lists the source for text from **INCLUDE** files as well as the original source file.

-m0

specifies that only 68000 instructions are allowed. This option issues warning messages if, for example, you use instructions that are only available on the 68020. This option is the default.

-m2

turns off the warnings generated by the **-m0** option. Use this option if 68020 instructions are allowed.

- m3**
turns off the warnings generated by the **-m0** option. Use this option if 68030 instructions are allowed.
- m4**
turns off the warnings generated by the **-m0** option. Use this option if 68040 instructions are allowed.
- oprefix**
specifies a prefix that you want placed in front of the output filename. If you specify a directory name as a prefix, include the (/). Any drive or directory prefixes in the input filename are discarded before the new prefix is added. Do not include blanks in the prefix.
- s**
includes the section name at the beginning of each hunk.
- u**
specifies that you want the assembler to prefix external references with an underscore (_). Do not use this option if you entered all external names with leading underscores. You should specify this option when you assemble startup code.
- w**
defines the symbol **shortint** to be 1. This option works as if you had specified **-dshortint=1**.

For example, the following command assembles the file named **modn.a**, produces an object file named **modn.o**, and generates a listing file named **modn.lst** that lists the source code and any error messages:

```
asm >modn.lst -l modn
```



Part 4

Using the SAS/C[®] Optimizer

Chapter 12 Optimizing Your Code

12 Optimizing Your Code

329	<i>Introduction</i>
329	<i>The Global Optimizer</i>
334	<i>The Peephole Optimizer</i>
334	<i>Running the Optimizers</i>
335	<i>Global Optimization Compiler Options</i>
336	<i>Global Optimization and the Debugger</i>

Introduction

Your SAS/C Development System provides the following two optimizers:

global optimizer (GO)

analyzes the intermediate code produced by the compiler, performs several types of optimizations, and produces output in the same form as the compiler. Because the global optimizer works on intermediate code, it has no knowledge of the target processor or its instructions.

peephole optimizer

analyzes the assembler instructions and replaces inefficient sequences of instructions with shorter, more efficient sequences.

The Global Optimizer

The global optimizer optimizes the flow of control and data through an entire function. The global optimizer performs several types of optimizations:

register assignment

analyzes the function to determine which auto variables, formal variables, temporary values, and constant values should be assigned to registers at each point in the function. The optimizer assigns up to three address registers, four data registers, and, if a math coprocessor is present, four floating-point registers for the use of register variables.

Generally speaking, the variables that are most used at a given point are assigned to registers. For example, variables occurring in

loops are more likely to be assigned to registers. The global optimizer attempts to keep a variable assigned to a register for as long as possible.

Using the ampersand (&) operator with a variable prevents the global optimizer from allocating that variable to a register because it cannot predict when the resultant pointer will be used to read or modify the variable's value in memory. The same condition applies to an external variable because it may appear in another function or the & operator may be used with it elsewhere.

The effect of global optimization's register allocation is quite different from the use of the `register` storage class. In general, a variable declared using the `register` keyword is associated with a machine register throughout the entire block in which it is declared (usually the entire function). In most functions, the variable is heavily used in some places and not used in other places. Yet, if a machine register is assigned to the variable, then the same register cannot be reused even in those sections where the variable is not used. Therefore, global optimization changes a register's assigned variable during the evaluation of the expression to ensure that the most heavily used variables are always in machine registers.

The global optimizer overrides the `register` keyword in the declarations of integer, double, and pointer variables. Because of the portability of the C language, it is difficult for a programmer to know the number of available registers provided by the target machine and the compiler. The concept of a register variable is based on the idea that the variable is kept in a register for the entirety of its scope. Such restrictions no longer apply when a compiler uses the more advanced registration allocation algorithms in SAS/C software. Even though the compiler does not have dynamic information about program execution that would indicate which statements are executed more heavily, it can use the loop nesting structure to make a reasonable approximation.

dead store elimination

eliminates the assignment of a value to a variable if the value is not used. The assignment can be eliminated as in this example:

```
index = 23;
.
.
.
/* code that does not refer to index */
.
.
.
index = 12;
```

The first assignment to `index` can be removed. Since the global optimizer inspects all references to the variable throughout the entire function, even subtle dead stores are eliminated.

dead code elimination

eliminates code that never can be executed.

common subexpression merging

eliminates recalculation of values that have been computed previously within the same function. For example, the following code

```
x = i / 3;
y = i / 3 + 4;
```

can be changed to

```
temp = i / 3;
x = temp;
y = temp + 4;
```

moving invariants out of loops

moves a calculation in a loop, whose value is the same on each iteration, to the outside of the loop. For example, the loop

```
for (i = 0; i < j; i++)
{
    a[i] = p->q.r[10];
}
```

can be changed to

```
temp = p->q.r[10];
for (i = 0; i < j; i++)
{
    a[i] = temp;
}
```

Refer to the explanation for `optloop` later in this chapter for more information about this type of optimization.

induction variable transformations

changes to addition loops containing multiplications (usually those associated with array indexing).

copy propagation

eliminates definitions of the form `leftvar = rightvar` when all uses of `leftvar` have this definition as the single reaching definition and `rightvar` will not change before each use. This optimization supports other optimizations.

constant propagation and folding

replaces references to a variable with a constant when the variable is defined as that constant. If the variable is used only in expressions with a different type (for example, if an `int` variable is only used in a comparison with `float` variables), global optimization creates a constant of the correct type. If the variable is used only as a constant, global optimization eliminates the variable entirely. The following example demonstrates these optimizations:

```
void f(double d)
{
    i = 10;
    for (; d < i; ++d)
    {
        .
        .
        .
    }
    return;
}
```

The previous code can be changed to the following:

```
void f(double d)
{
    for (; d < 10.0; ++d)
    {
        .
        .
        .
    }
    return;
}
```

Constant propagation is often useful in programs that contain inline functions since one or more parameters to the inlined function actually may be constants.

auto variable elimination and remapping

eliminates unused auto variables and reassigns storage offsets. Often the variable is unused because of previous optimizations.

very busy expression hoisting

moves an expression that is computed along all paths from a point in the code to a single, common location. For example, the code

```
if (expression)
    x = i + j;
else
    y = (i + j) * 2;
```

can be changed to

```
temp = i + j;
if (expression)
    x = temp;
else
    y = temp * 2;
```

various reductions in strength

performs associative reordering of additive operations involving constants to reduce the operation count.

Various arithmetic operations involving constants are reduced in strength.

Conditional and logical expressions whose results are unused are converted into corresponding **if** code. For instance, **putchar** from **stdio.h** is implemented with a conditional expression. If the result (the original character or an error indication) is not used, GO converts it into **if else** code, eliminating a load into a register.

various control flow transformations

performs various transformations to eliminate unreachable code or useless control structures.

reordering of operations to reduce value lifetimes

moves expressions with a single use adjacent to the operation that uses them. This optimization helps reduce temporary lifetimes and supports optimizations that move code around. For example, in the following code, the computation of the address **&p[i]** can be moved after the call:

```
p[i] = f();
```

The Peephole Optimizer

The code generator contains all of the knowledge about the target processor and its instructions and makes full use of the 680x0 instruction set. The code generator tries not to generate extra instructions, but the peephole optimizer can help catch the few places where this is not possible.

As stated earlier, the peephole optimizer analyzes the assembler instructions and replaces inefficient sequences of instructions with shorter, more efficient sequences. For example, for the following set of instructions, the peephole optimizer deletes the `TST.L` instruction:

```
MOVE.L    D1,D2
TST.L     D2
BNE       LABEL
```

The test instruction is not needed because the condition codes for the branch instruction are set by the move instruction.

Currently, the peephole optimizer optimizes 22 patterns of assembler instructions. If you find additional patterns that the optimizer should recognize, contact the Technical Support Division at SAS Institute Inc.

Running the Optimizers

To run the optimizers, specify the appropriate compiler option:

Option	Effect
<code>optimize</code>	turns on both the global and peephole optimizers.
<code>optimize</code>	turns on only the global optimizer.
<code>nooptpeep</code>	

In many cases, optimized code is more difficult to debug than non-optimized code. You may want to use the optimizer only after the main program has been tested and most errors have been corrected.

Global Optimization Compiler Options

The compiler accepts the following options to modify the operation of the global optimizer:

optalias

disables type-based aliasing assumptions. If **optalias** is used, the global optimizer uses worst-case aliasing. Using this option can significantly reduce the amount of optimization that can be performed. The **nooptalias** option is the default.

optcomplexity

defines the complexity of functions considered small by **optinline**. For more information, refer to Chapter 8, “Compiling and Linking Your Program,” of *SAS/C Development System User’s Guide, Volume I: Introduction, Compiler, Editor*.

optdepth

defines the maximum depth of function calls to be inlined. The range is 0 to 6, and the default value is 3.

optinline

inlines small functions (as defined by **optcomplexity**) as well as those with the `__inline` keyword. The **optinline** option is the default when **optimize** is used. Refer to Chapter 11, “Using Amiga Specific Features of the SAS/C Language,” in the *SAS/C Development System User’s Guide, Volume I*, for information about the `__ts__inline` keyword.

optinlocal

inlines single-call static (local) functions.

optloop

assumes that loops have multiple iterations when the number of iterations is variable. This assumption enables the movement of safe code out of loops. The **optloop** option is the default. For more information, see “moving invariants out of loops,” earlier in this chapter.

When a loop is not executed at all, the moved code is executed in cases where it previously would not have been. For example, the code

```
for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
        p[i * m + j] += 1;
```

can be changed to

```
for (i = 0; i < n; ++i)
{
    temp = i * m;
    for (j = 0; j < m; ++j)
        p[temp + j] += 1;
}
```

In the changed code, `i*m` can be calculated when `m` is less than or equal to 0. When `optloop` has been specified, there may be a small cost in time for every loop that is not executed. There is also a significant time saving for loops that are executed many times, as most are.

Some types of code may cause an exception, for example, division by 0. For this reason, the global optimizer restricts moved code to safe operations, including integral and pointer arithmetic other than division by 0, but not including floating-point operations. The global optimizer avoids incorrect exceptions regardless of the setting of `optloop`.

optrdepth

defines the maximum level of recursive function calls to be inlined. The range is 0 to 6, and the default is 1.

optsize

disables optimizations that might sacrifice code size to save time. Only some loop optimizations do this. The `nooptsize` option is the default. Specifying both `opttime` and `optsize` may result in code that is both slower and larger than most optimized code.

opttime

disables optimizations that might sacrifice time to save space. The `noopttime` option is the default.

Global Optimization and the Debugger

To use all the capabilities of the debugger, the lines in the source code file must correspond exactly with the lines in the object code file. Optimizing your code may change this correspondence. Also, the debugger may not know about variables that the optimizer moves or eliminates.

If you use the debugger on optimized code, run the debugger in *mixed mode* by entering the following command on the debugger command line:

```
opt source mixed
```

In mixed mode, the debugger displays both assembly language instructions and C source lines in the source window. Use the assembly language code to determine if the variables you examine are actually located where the debugger thinks they are.

■ Part 5

Appendix

Appendix

diff File-Matching Algorithm



Appendix

diff File-Matching Algorithm

This appendix describes the file-matching algorithm used by the `diff` utility. The algorithm used by the `diff` utility is described in *A Technique for Isolating Differences Between Files* by P. Heckel.*

The matching algorithm described by Heckel is simple, fast, and effective. Briefly, it works as follows:

- Identify those lines that occur only once in each file. Assume that these lines match one another.
- Sweep downward into the first file. Each time you come to a line L that is already matched to the line L^1 in the second file, examine the next line. This second line is referred to as $L+1$.
- If $L+1$ is the same as the corresponding line L^1+1 in the second file, then match those lines as well.
- After this downward sweep is completed, sweep upward in the first file attempting to match lines in the same way (except that you look at $L-1$ and L^1-1 when finding a matched line L).

The effect of the matching algorithm is to match blocks of text to each other in the two files.

However, the algorithm frequently cross-matches single lines or two blocks of lines. That is, lines L and $L+k$ may be matched to lines L^1 and L^1-j . Consequently, after the process described previously is completed, the matches must be untangled to generate a coherent report of the differences. Therefore, the `diff` utility cross-matches blocks and then unmatches the smaller of the blocks. This process is an attempt to satisfy the notion that the differences reported should be minimal.

In addition, this algorithm must first seed itself by identifying lines that occur exactly once in each file. Therefore, this algorithm may fail to match any lines even when there is an obvious match.

* Heckel, P. (1978), "A Technique for Isolating Differences Between Files," *Communications of the ACM* (April), 264-8.

For example, you may have two files with the following contents:

File 1	File 2
aaa	bbb
xxx	xxx
xxx	xxx
yyy	zzz

The algorithm will fail to match the blocks of **xxx** lines because it does not find any lines that occur exactly once in each file to seed the algorithm.

Index

A

- a option, smake utility 289
- abort command, SC_SCMSG AREXX port 262, 262–264
- absolute addressing modes 313
- absolute expressions 304
- accelerator keys 23–24, 29, 34
- actions, smake utility 273
- actions gadget 267
- activate command 107, 111, 127
- Address field, tasks command 109
- address parameter 118–119
 - examples 118–119
 - using with location parameter 40
- addresses
 - absolute addressing modes 313
 - addressing modes supported by SAS/C
 - assembler 312
 - dumping 62
 - forcing program counter-relative
 - addressing 311
 - register-direct addressing modes 313
 - specifying with CSECT directive 311–314
 - specifying with SECTION directive 309–310
- addsym option
 - adding to slink automatically 8, 9
 - generating H_SYMBOL hunk 8
 - purpose 74
- AddTask function 114
- after option
 - break command 49, 135
 - go command 40, 42, 156
- alias command 89–92, 128–129
 - See also define command
 - See also expand command
 - compared with define command 92, 93
 - displaying alias definitions 90
 - displaying current aliases 89–90
 - escape characters 19
 - examples 91, 128–129
 - expanding alias names 90, 91
 - nesting aliases 92
 - positional parameters 90
 - referencing aliases commands with
 - back-tick character (`) 92
 - syntax 89, 128
- aliases for dump command in previous releases 146–147
- alternate targets, smake utility 283–284
- altfile command, SC_SCMSG AREXX port 262
- altline command, SC_SCMSG AREXX port 262
- ampersand operator (&)
 - identifying addresses 118
 - using with global optimizer 330
- ampersand symbol (&)
 - special symbol in smake utility 277
- and operator (&)
 - not used with register variables 58
- append block, diff utility 221
- AREXX interface 101
 - invoking editors 261–262
- AREXX macros 101–104
 - command output 103–104
 - invoking macros 102–103
 - libraries.cpr 106
 - macros provided by CodeProbe 101–102
 - return codes 103
 - values returned from CodeProbe
 - commands 103–104
- AREXX port 262–264
- args command 130
- array-slice parameter 119–120
- arrays
 - displaying 31, 59–61
 - setting watches or watch breaks 71
- arrdim option, opt command 86–87, 165
- ASCII option, dump command 65–66
- asm command 323–326
 - c option 324
 - d option 324
 - d symbol(=value) option 324
 - h option 324–325

asm command (*continued*)

- i option 325
- m0 option 325
- m2 option 325
- m3 option 326
- m4 option 326
- o option 326
- options supported 324–326
- s option 326
- syntax 323
- u option 326
- w option 326
- _asm keyword 319–320
- assembler directives 304–308
 - See also specific directives
- assembly language 299–326
 - addresses, specifying with CSECT
 - directive 311–314
 - assembler directives 304–308
 - assembler statements, writing 300–304
 - _asm keyword 319–320
 - calling assembler functions from C
 - functions 314–321
 - calling C functions from assembler
 - functions 322
 - communicating with C language 314–323
 - control sections, defining 309–314
 - CSECT directive 310–314
 - expressions, specifying 302–304
 - fields in source line 300–301
 - format of source lines 300
 - macros, defining and using 308
 - options supported by assembler 324–326
 - parameters=register option 318–319
 - passing parameters on the stack 315–318
 - referencing global data 322–323
 - returning values to calling function
 - 320–321
 - running the assembler 323–326
 - SECTION directive 309–310
 - size suffixes, specifying 301–302
 - writing assembly language functions
 - 300–314
- assembly mode
 - ps command 47
 - setting 46, 83
 - trace command 45
- asterisk (*)
 - grep search pattern 225, 226, 230–231
 - removing all aliases 92

- used in array-slice parameter 119
- wild card indicator 34
- asynchronous tasks 108
- at sign (@)
 - oml utility 251, 253
 - replacing underscore in function names
 - for compiler 318
 - special symbol in smake utility 277
 - specifying octal numbers in assembler
 - statement expressions 303
- auto variable elimination and remapping
 - 333
- autoedit option, scmsg utility 255
- autoswap mode 29, 88
- autoswap option, opt command 88, 165

B

- b option
 - diff utility 219
 - oml utility 253
 - smake utility 289
- back-tick character (`)
 - non-expansion of escaped macro
 - definitions 92–93, 141
 - referencing aliased commands 92
- backslash character (\)
 - continuing lines in aliases 90
 - Control-\ for terminating oml 251
 - escape character in grep searches 227, 231, 232–233
 - escape character preceding semicolon 93
 - extending commands across lines 18
 - interpretation in character strings 19, 123
 - special symbol in smake utility 277
- badchar option, opt command 87, 165
- bclear command 131
 - definition 36, 48
 - examples 131
 - syntax 131
- bdisable command 48, 132
- benable command 48, 133
- blist command 36, 48, 134
- bottom command, SC_SCMSG AREXX
 - port 262
- braces ({})
 - defining aliases 90
 - using in cmd_list parameters 50

brackets ([])
 enclosing characters in grep search 226, 228–230

branch instructions
 assembler statement size suffixes 302

break command 48–53, 135–136
 See also bclear command
 See also bdisable command
 See also benable command
 See also blist command
 after option 49, 135
 cmd_list parameter 50, 135
 definition 39
 examples 136
 go command as last command of cmd_list parameter 51
 quiet option 52–53, 135
 syntax 48–49, 135
 temp option 52–53, 135
 trace option 51, 52–53, 135
 when option 49, 135

Break menu 21–22

breakpoints
 attaching actions to 50–53
 caution against placing in libraries 110
 conditional 49–50
 definition 5, 32, 47
 executing up to a breakpoint 32
 multitasking programs 108
 nesting 52
 preventing debugger from stopping 51
 resident libraries 105, 106
 setting 32, 47–48
 temporary, set by go command 40, 43
 turning off 32
 using with detach command 111–112

-buffer command-line option 78–79

build command, SC_SCMSG AREXX port 262

Build menu, scmsh utility 259

built-in functions 202–203
 See also specific functions
 list of functions 203
 parameters 202
 purpose 202

C

C functions, calling
 See calling assembler functions from C functions
 See calling C functions from assembler functions

C mode 83

-c option
 asm command 324
 diff utility 219
 grep utility 225
 smake utility 289

call command 137–138

call frame 6

calling assembler functions from C functions 314–321
 —_asm keyword 319–320
 calling an assembler module 316
 parameters=register option 318–319
 passing parameters on the stack 315–318
 retrieving parameters from the stack 317
 returning values to calling function 320–321
 saving register contents 317–318
 stack, example 317

calling C functions from assembler functions 322

Calls window 14

caret (^)
 grep search pattern 226

case option, opt command 85, 165

catch command 107, 112, 139

Catch New Devices option
 See devices option, opt command

catch option, opt command 89, 166

change block, diff utility 221

character classes, grep utility 226
 specifying 228–230

characters and strings, dumping 65–66

class command, SC_SCMSG AREXX port 262

clear command, SC_SCMSG AREXX port 262

-cli command-line option 11, 79

cmd_list parameter, break command 50, 135

CNOP assembler directive 304

Code Generation Options window 266

CodeProbe 3–24

See also debugging programs

basic features 4

changes and enhancements 5

command line 16–19

compiling and linking programs 9

controlling 8

cpr command 9–10

debugging procedure 7–8

entering commands 16–24

function and special keys 22–23

introduction 3–4

line mode 11

menu accelerator keys 23–24

multitasking features 4–5

opening windows 15–16

pull-down menus 20–22

quit command 10

requirements 7

running 9–11

running on optimized code 336–337

starting 25–34

terminology 5–7

windowing interface 12–16

Workbench support 10–11

CodeProbe sample session 25–34

activating autoswap mode 29

breakpoints, setting 32

clearing watches and watch breaks 34

compiling and linking 25

continuing execution 30

controlling debugging session 27–34

displaying arrays and structures 30–32

displaying variable values and types 32

executing breakpoints 32

executing until variable changes value

33–34

go command 26

invoking the debugger 26

online help 26–27

quitting the debugger 34

restart command 26

source mode, setting 27–28

starting the debugger 25–26

stepping over code 29–30

watch and watch break commands 32–34

comma (,)

separating multiple arguments in display
command 58

-command command-line option 79

command-line editing 16–19

determining editing modes 17

editing function keys 17–18

escape characters in commands 19

escape sequences in character strings
18–19

extending commands across lines using
backslash (\) 18

command-line options 78–81

adding as tool types 78

-buffer 78–79

-cli 11, 79

-command 79

-device 97

-i 79

-line 79

-nok 98

-noprofile 79–80

-pipe 97

-screen 80

setting up in Workbench screen 11

-speed 97

-startup 26, 80

-symfile 98

-unit 97

using with cpr command 10

-w 80

-wb 11, 80

-wdialog 81

-wregister 81

-wsource 81

-wwatch 81

command output for AREXX macros

103–104

command parameter, help command 27

commands

See also specific commands

address parameter 118–119

array-slice parameter 119

command-line editing 16–19

command-line equivalents for menu items
20–22

customizing Dialog window commands
89–94

editing function keys 17–18

editing modes 17

entering on command line 16–19

escape characters in strings 18–19

expression parameter 120

- extending across lines using backslash (\) 18
 - function and special keys 22–23
 - list of commands 116–118
 - location parameter 120–121
 - menu accelerator keys 23–24
 - number parameter 121
 - pull-down menus 20–22
 - range parameter 121–122
 - register parameter 122–123
 - SC_SCMSG AREXX port 262–264
 - shortcuts for command names 16
 - special parameters 118–125
 - specifying in oml utility 252–253
 - string parameter 123
 - subrange parameter 123–124
 - syntax 16
 - type parameter 124
 - variable parameter 125
 - > prompt 16
- comment field, assembly language source line 301
- comment lines, cprinit file 82
- communications parameters
 - defining for cross debugging 97
 - device 97
 - pipe 97
 - speed 97
 - unit 97
- comparing files
 - See diff utility
- compiler
 - See sc command
- compiler options, setting
 - See scopts utility
- Compiler Options Index window 266
- Compiler Options window 266
- compiling
 - compiling and linking programs 7
 - sample program (lines.c) 25
- condition control register (CCR) flags 110
- config= option, scmsg utility 255
- constants
 - assembler statement expressions 303
 - optimizing 332, 333
 - reductions in strength 333
- context option, opt command 85, 166
- control flow transformations 333
- control sections
 - See CSECT assembler directive
 - See SECTION assembler directive
- controlling program execution
 - See program execution, controlling
- copy propagation 332
- cover utility 216–218
 - dir= option 217
 - example 217–218
 - merge option 217
 - nosource option 217
 - options supported 217
 - running 216
 - syntax and description 216–217
- cpr command 9–10
 - available command-line options 10
 - changing the default environment 11
 - cli option 11
 - invoking the debugger 26
 - syntax 9, 75
 - wb option 11
- .cpr extension for macros 102
- cprinit file
 - comment lines 82
 - examples 81–82
 - initializing CodeProbe 81–82
 - suppressing execution with -noprofile option 79–80
- CPRK
 - See also cross-development mode
 - examples 99
 - options 98
 - required on target machine 96
 - running 96
 - starting before CPRX 97, 98
 - starting the kernel 98
 - terminating 99–100
- CPRX
 - See also cross-development mode
 - examples 99
 - nok option 98
 - starting 98–99
 - stripping debug information with -x option 96
 - symfile option 98
 - syntax 98
 - terminating 99–100
 - x option 99
- cross-debugging 6
- cross-development mode 95–100
 - See also CPRK
 - See also CPRX
 - debugging applications 99

- cross-development mode (*continued*)
 - defining communications parameters 97
 - examples 99
 - host machine 96
 - NULL modem cable 96
 - preparing to use cross debugger 96
 - procedure 96–99
 - reasons for cross-debugging 95–96
 - running cross debugger 96–100
 - starting the cross debugger (CPRX) 98–99
 - starting the kernel (CPRK) 98
 - target machine 96
 - terminating CPRX and CPRK 99–100
 - CSECT assembler directive 310–314
 - addressing modes supported 312
 - definition 304
 - forcing program counter-relative addressing 311
 - format 310
 - parameters 310
 - program counter-relative addressing modes, examples 314
 - register-direct addressing modes 313
 - specifying addresses 311–314
 - specifying immediate data 314
 - valid types and sizes of relocation information 311
 - CTRL-C, terminating CPRK and CPRX 99
 - Ctrl-L key 23
 - Ctrl-W key 23
 - curly braces
 - See braces (| |)
 - current task, displaying 89
 - cycles gadget 267
- D**
- d command
 - oml utility 252
 - short form of display command 30
 - d option
 - asm command 324
 - smake utility 289
 - splat utility 291
 - d symbol(=value) option, asm command 324
 - data, examining
 - See dump command
 - data structures, displaying
 - See structures, displaying
 - DC assembler directive 305
 - DCB assembler directive 305
 - deactivate command 107, 111, 140
 - dead code elimination 331
 - dead store elimination 330
 - Debug field, tasks command 109
 - DEBUG tool 11
 - debug= options 74–76
 - determining which option to use 76–77
 - list of options 74
 - debug=full option 74
 - using 76
 - when to use 76–77
 - debug=fullflush option 74
 - using 76
 - when to use 76–77
 - debug=line option 74
 - generating H_DEBUG hunks 8
 - using 75
 - debug=symbol option 74
 - generating H_DEBUG hunks 8
 - using 75–76
 - when to use 76–77
 - debug=symbolflush option 74
 - adding to sc command 9
 - displaying full debugging information 25
 - using 76
 - when to use 76–77
 - debugger
 - See CodeProbe
 - debugging environment, setting up 73–94
 - alias command 89–92
 - arrdim option 86–87
 - autoswap mode 88
 - badchar option 87
 - case option 85
 - Catch New Devices option 88
 - catch option 89
 - changing default environment with -cli and -wb options 11
 - choosing command-line options 78–81
 - context option 85
 - current task 89
 - customizing Dialog window commands 89–94
 - debug= options 74–76
 - debug=full option 76
 - debug=fullflush option 76

- debug=line option 75
- debug=symbol option 75–76
- debug=symbolflush option 76
- define command 92–93
- echo mode 84
- ibytes option 84–85
- ignorepath option 85
- initializing CodeProbe 81–82
- list option 86
- radix option 86
- rangelen option 87
- search path 87–88
- setting and showing options 82–89
- source file location 77–78
- source modes 83–84
- Step Into ResLib option 88–89
- strlen option 86
- unalias command 92
- unassemble option 86
- undefine command 94
- when to use debug= options 76–77
- debugging programs 25–34, 35–72
 - See also multitasking programs, debugging
 - See also resident libraries, debugging
 - activating autoswap mode 29
 - break command 48–53
 - breakpoints 47–53
 - clearing watches and watch breaks 34
 - commands for running sample programs 36
 - compiling and linking programs 9, 25
 - continuing execution 30
 - controlling program execution 27–34, 39–53
 - display command 57–61
 - displaying arrays and structures 30–32
 - displaying variable values and types 32
 - dump command 61–66
 - examining data and data structures 57–72
 - executing breakpoints 32
 - executing until variable changes value 33–34
 - go command 26, 39–44
 - invoking the debugger 26
 - online help 26–27
 - proceed commands 46–47
 - quitting the debugger 34
 - register command 66–68
 - restart command 26
 - running CodeProbe 9–11
 - sample program 1 37–38
 - sample program 2 53–56
 - setting breakpoints 32
 - setting source mode 27–28
 - single stepping 44–47
 - starting the debugger 25–26
 - stepping over code 29–30
 - steps for debugging 8
 - trace commands 45–46
 - watch and wbreak commands 32–34, 70–72
 - watches and watch breaks 69–72
 - whatis command 68–69
- decimal option, dump command 65
- .DEFAULT fake target 281–282
- default files, smake utility 287–288
- #define capability, C preprocessor 93
- define command 141
 - compared with alias command 92, 93
 - compared with C preprocessor #define capability 93
 - defining symbols to nothing 93
 - displaying macro definitions 93
 - examples 141
 - syntax 93, 141
- delcomp command, SC_SCMSG AREXX port 262
- delete block, diff utility 220
- delete command, SC_SCMSG AREXX port 263
- delfile command, SC_SCMSG AREXX port 263
- delnum command, SC_SCMSG AREXX port 263
- dependent file, smake utility 273
- detach command 111–112, 142
 - definition 107
 - examples 142
 - precautions 111–112
 - syntax 111, 142
- device command-line option 97
- devices option, opt command 88, 166
- Diagnostic Message Options window 266
- Dialog window
 - echo mode 84
 - functions 14
 - open by default 12, 26
 - scrolling 31

Dialog window (*continued*)
 specifying start-up window coordinates 81
 zooming to full size 30

Dialog window commands
 alias 89–92
 customizing 89–94
 wclear 34

diff file-matching algorithm 341–342

diff utility 219–223
 append block 221
 -b option 219
 -c option 219
 change block 221
 delete block 220
 error messages 222–223
 examples 221–222
 -F option 219
 -L option 219
 -l option 219
 -o option 219
 options supported 219–220
 -p option 220
 -q option 220
 syntax and description 219–221
 -w option 220

dir= option, cover utility 217

disassembling object modules (omd utility) 249

display command 57–61, 143–144
 compared with register command 77
 examples 143–144
 expression parameter 57
 format parameter 57
 multiple arguments 58
 overriding default format 57
 string parameter 58
 structures and arrays 30–32, 59–61
 syntax 57, 143
 value and type of variables 32
 variable types 75

dollar parameter (\$)
 dump command 145
 line command 161
 set command 177

dollar sign (\$)
 grep search pattern 227, 229, 231
 indicating macros 277, 278
 positional parameters in aliases 90, 128
 prefixing register names 119

 special symbol in smake utility 277
 specifying hexadecimal numbers in assembler statement expressions 303
 -\$ option, grep utility 225

DS assembler directive 305

dump command 61–66, 145–147
 addresses, dumping 62
 ASCII option 65–66
 characters and strings, dumping 65–66
 compared with register command 77
 compatibility aliases 146–147
 decimal option 65
 examples 147
 float option 64
 floating-point numbers, dumping 64–65
 format parameter 145
 integers, dumping 65
 pointers, dumping 63–64
 ranges, dumping 62–63
 syntax 61, 145
 variables, dumping 61–62, 75

dzero command 148
 displaying ASCII text string 66
 examples 31, 148
 syntax 148

E

-e option, smake utility 289

echo command 149

echo option, opt command 84, 166

Edit menu, scmsg utility 260
 AREXX for invoking editors 261–262
 examples 264–265
 SC_SCMSG AREXX port 262–264

editcommand option, scmsg Project menu 257

editing modes 17

ELSE assembler directive 305

END assembler directive 305

ENDC assembler directive 305

ENDM assembler directive 305, 307

env command 150–151

environment 6
 See also debugging environment, setting up
 representing state of the machine 14
 run environment 6
 user environment 6

environment variable (sc/cprpath) 11
 EQU assembler directive 305
 equals sign (=)
 defining macros in smake utility 277–278
 EQUR assembler directive 305
 errnum command, SC_SCMSG AREXX
 port 263
 errors and warnings, searching for
 See scmsg utility
 escape characters 18–19
 alias command 19
 backslash (\) in character strings 19
 backslash (\) in grep searches 227
 character strings 18–19
 commands 19
 double quotes (“ ”) around 18–19
 dumping with ASCII option 66
 single quotes (‘ ’) around 19
 string parameter support 123
 exception handling 113–114
 exclamation mark (!)
 marking watch breaks 33
 negation character in grep search 226,
 230
 execute command 152
 executing code
 See program execution, controlling
 EXITM assembler directive 305, 307
 expand command 153
 expression parameter
 display command 57
 operators not accepted 120
 expressions
 absolute 304
 optimizing subexpressions 331
 reductions in strength 333
 relocatable 304
 reordering 333
 specifying in assembler statements
 302–304
 very busy expression hoisting 333

F

-f option
 grep utility 225
 lstat utility 244
 smake utility 289
 -F option, diff utility 219

FAIL assembler directive 305
 fake targets, smake utility 281–283
 file command, SC_SCMSG AREXX port
 263
 file comparison
 See diff utility
 file dependencies, tracking
 See smake utility
 file-matching algorithm 341–342
 File menu
 Line and Find Current Line items 20
 menu items and command-line
 equivalents 20–22
 finish command 100, 154
 flags, modifying
 See rflag command
 float option, dump command 64
 floating-point instructions
 assembler statement size suffixes 301
 floating-point numbers
 dumping 64–65
 hexadecimal notation in assembler
 statement expressions 303
 FORMAT assembler directive 305
 format parameter
 display command 57
 dump command 145
 register command 155
 function keys
 actions performed 22–23
 command-line editing 17–18
 function line parameter, location
 parameter 41
 function parameter, location parameter 41,
 42
 functions
 See built-in functions

G

gadgets in scopts utility 267–268
 global data, referencing 322–323
 global optimizer 329–333
 auto variable elimination and
 remapping 332
 common subexpression merging 331
 constant propagation and folding 332
 control flow transformations 333
 copy propagation 332

global optimizer (*continued*)

- dead code elimination 331
- dead store elimination 330–331
- induction variable transformations 331
- moving invariants out of loops 331
- optalias option 335
- optcomplexity option 335
- optdepth option 335
- optinline option 335
- optinlocal option 335
- optloop option 335–336
- optrdepth option 336
- optsize option 336
- opttime option 336
- reductions in strength 333
- register assignment 329–330
- reordering of operations 333
- running 333–337
- running the debugger on optimized code 336–337
- very busy expression hoisting 333

global symbol information 8

global symbol tables

- See *gst* utility
- See *hypergst* utility

go command 39–44, 156–157

- after clause 42, 156
- completing execution before running other commands 30
- examples 44, 156–157
- executing until breakpoint 30, 32
- last command of *cmd_list* parameter 51
- location parameter 156
- location parameter forms 40–42
- parameters 40
- starting execution of sample program 26
- syntax 39
- syntax errors 43–44
- when clause 42–44, 156

go main command 26, 80

gotofile option, *scmsg* Project menu 257

gotoline option, *scmsg* Project menu 258

greater than sign (>)

- See also *redirecting output* (>)
- > prompt for command line 16

grep utility 224–236

- c option 225
- description 224–227
- error messages 234–236
- examples 227

- f option 225
- n option 225
- number of times characters may appear, specifying 230–231
- options supported 225
- p option 225
- patterns at beginning or end of line, specifying 231
- patterns containing special characters, specifying 232–234
- q option 225
- redirecting output 224
- s option 225
- single character, specifying 228
- specifying patterns 225–227
- syntax 224
- v option 225
- V option 225
- \$ option 225

gst utility 237–239

- error messages 239
- examples 238–239
- list option 237–238
- options supported 237–238
- syntax and description 237–238
- unload option 238
- verbose option 238

H

- h option
 - asm* command 324–325
 - smake* utility 290
- H_DEBUG* hunks 7, 8
- help* command 26–27, 158
- Help* key 23
- Help* menu 16
- Help* window 14
- hidden option, *scmsg* Project menu 258
- hidden option, *scmsg* utility 255
- hide* command, *SC_SCMSG AREXX* port 263
- host machine 6, 96
- hunks
 - H_DEBUG* hunks 7, 8
 - H_SYMBOL* hunks 8
- hunks* command 159
- hypergst* utility 240–241
- hyphen (-)
 - specifying range in *grep* search 226

I

- i command-line option 79
- i option
 - asm command 325
 - smake utility 290
- ibytes option, opt command 84–85, 166
- IDNT assembler directive 305
- IF assembler directive 305
- IFC assembler directive 305
- IFD assembler directive 306
- IFEQ assembler directive 306
- IFGE assembler directive 306
- IFGT assembler directive 306
- IFLE assembler directive 306
- IFLT assembler directive 306
- IFNC assembler directive 306
- IFND assembler directive 306
- IFNE assembler directive 306
- .IGNORE fake target 282
- ignorepath option, opt command 85, 166
- image 6
- image parameter, location parameter 42
- INCLUDE assembler directive 306
- infinite loops 112
- initializing CodeProbe
 - See cprinit file
- Input window 15
- insert editing mode 17
- instruction bytes
 - See ibytes option, opt command
- integers, dumping 65
- interlace mode
 - See -i command-line option
- invoking the debugger
 - See cpr command

J

- jump command 89, 160

K

- k option, smake utility 290
- kernel
 - See CPRK

keys

- function key actions 22–23
- function keys for command-line editing 17–18
- menu accelerator keys 23–24, 29, 34
- special keys 23

L

- l command, oml utility 252
- l option
 - diff utility 219
 - tb utility 295
- L option, diff utility 219
- label field, assembly language source line 300
- left-Amiga-m keyboard shortcut 29, 34
- less than sign (<)
 - diff utility 220
 - oml utility 251
- libraries
 - See resident libraries, debugging
- libraries, managing
 - See oml utility
- libraries.cpr AREXX macro 106
- line boundary 6
- line command, SC_SCMSG AREXX port 263
- line command-line option 11, 79
- line parameter, location parameter 41
- link library 250
- link option 9, 25
- linker
 - See slink
- Linker Options window 267
- linking programs 9, 25
- LIST assembler directive 306
- list command 161–162
- list option
 - gst utility 237
 - opt command 86, 166
- Listing/Cross Reference Options window 266
- lists gadget 267
- LLEN assembler directive 306
- load module 324
- local input files, smake utility 284–286

location parameter 40–42, 123–124
 address parameter 40
 definition 40, 120
 forms 40
 function line parameter 41
 function parameter 41, 42, 121
 go command 40–42, 156
 image parameter 42, 120
 integer parameter 121
 line parameter 41
 module parameter 42, 120
 syntax 120

log command 163–164

log file 6

loops

induction variable transformations 331
 moving invariants out of loops 331

lprof utility 242–243

lstat utility 244–248

-f option 244

options supported 244

syntax and description 244

-t option 244

-z option 244

M

-m option, tb utility 295

MACRO assembler directive 306

macros

See also AREXX macros

See also define command

defining assembly language macros 308

escaped macros not expanded 92–93

hiding in double quotes 92

macros, smake utility 277–281

default macros 279–280

defining transformation rules 280–281

defining with equals sign (=) 277–278

overriding definitions 278

referring to macros 278

Map Options window 267

MASK2 assembler directive 306

maximum bad characters

See badchar option, opt command

memcmp function 204

memcpy function 205

memmove function 206

Memory menu 22

Memory window 15

memory, dumping

See dump command

memset function 207

menu accelerator keys 23–24, 29, 34

menus

See pull-down menus

merge option, cover utility 217

Message window 15

See also wmsg command

MEXIT assembler directive 306

minus sign (–)

special symbol in smake utility 229, 277

mixed mode 83–84

opt source mixed command 46, 77

running the debugger on optimized code
 336–337

setting 83–84

module 7

module parameter, location parameter 42

Modules window 15

multiple targets, smake utility 283

multitasking programs, debugging 107–114

activate command 111

asynchronous tasks 108

breakpoints and task handling 108

catch command 112

CodeProbe features 4–5

commands for controlling tasks 107,
 109–113

deactivate command 111

design considerations 113–114

detach command 111–112

exception handling 113–114

opt task command 110–111

symload command 112–113

synchronous tasks 108

task handling by CodeProbe 108

tasks command 109–110

terminating tasks 108

trap handling 113–114

types of tasks 108

-m0 option, asm command 325

-m2 option, asm command 325

-m3 option, asm command 326

-m4 option, asm command 326

N

- n option
 - grep utility 225
 - oml utility 253
 - smake utility 290
- Name field, tasks command 109
- name= option, gst utility 238
- NARG assembler directive 306
- nested breakpoints 52
- networks
 - using for cross-debugging 97
- next command, SC_SCMSG AREXX port 263
- nodebug option 74
- NOFORMAT assembler directive 306
- nok option 98
- NOLIST assembler directive 307
- non-floating-point instructions
 - assembler statement size suffixes 301
- NOOBJ assembler directive 307
- NOPAGE assembler directive 307
- noprofile command-line option 79–80
- nosource option, cover utility 217
- NULL modem cable for cross debugging 96
- number parameter 121
- numeric keypad modes 17

O

- o option
 - asm command 326
 - diff utility 219
 - oml utility 253
 - splat utility 291
- Object Module Disassembler 249
- Object Module Librarian
 - See oml utility
- objects, determining type or configuration
 - See whatis command
- OFFSET assembler directive 307
- omd utility 249
- oml utility 250–254
 - b option 253
 - commands, specifying 252–253
 - d command 252
 - examples 254
 - invoking 250
 - l command 252

- n option 253
- o option 253
- options, specifying 253
- r command 252
- s option 253
- syntax and description 250–251
- t option 253
- terminating 251
- v option 253
- x command 252
- x option 253
- @ command 253
- .ONEERROR fake target 282
- online help
 - See help command
- OpenDevice function 114
- OpenLibrary function 114
- operand field, assembly language source line 300
- operation field, assembly language source line 300
- operators
 - assembler statement expressions 303–304
- OPSYN assembler directive 307
- opt command 82–89, 165–167
 - arrdim option 86–87, 165
 - autoswap option 88, 165
 - badchar option 87, 165
 - case option 85, 165
 - catch option 89, 166
 - context option 85, 166
 - devices option 88, 166
 - displaying current task 89
 - echo option 84, 166
 - env option 112
 - examples 167
 - ibytes option 84–85, 166
 - ignorepath option 85, 166
 - list option 86, 166
 - radix option 86, 167
 - rangelen option 87, 167
 - reslib option 88–89, 167
 - search option 87–88, 166
 - source option 46, 77, 83, 84, 166, 336
 - strlen option 86, 167
 - syntax 165
 - task option 107, 110–111, 167
 - unassemble option 86, 167
- opt env command 112

opt source asm command 46
 opt source mixed command 46, 77, 84, 336
 opt task command 110–111
 definition 107
 examples 111
 task addresses 111
 task-ID argument 110
 using 110–111
 optalias option, global optimizer 335
 optcomplexity option, global optimizer 335
 optdepth option, global optimizer 335
 optimized code, running debugger on 336–337
 Optimizer Options window 266
 optimizing code
 See global optimizer
 See peephole optimizer
 optinline option, global optimizer 335
 optinlocal option, global optimizer 335
 option results command (AREXX) 103
 options
 See also command-line options
 See also debug= options
 See also opt command
 See also scopts utility
 addsym 8, 9, 74
 cover utility options 217
 default options, displaying 82
 diff utility options 219–220
 global optimizer compiler options 335–336
 grep utility options 225
 gst utility options 237–238
 lstat utility options 244
 oml utility options 253
 parameters=register option 318–319
 scmsg utility options 255–256
 scmsg utility Project menu options 257–258
 setting and showing within CodeProbe 82–89
 smake utility 289–290
 splat utility 291
 tb utility 295
 Options menu 20–21
 optloop option, global optimizer 335–336
 optrdepth option, global optimizer 336
 optsize option, global optimizer 336
 opttime option, global optimizer 336

Output window 15
 overwrite editing mode 17

P

-p option
 diff utility 220
 grep utility 225
 smake utility 290
 PAGE assembler directive 307
 parameters=register option 318–319
 pass count 7, 40, 42
 specifying 135
 patterns, searching for
 See grep utility
 pausing 179
 peephole optimizer 329
 functions 334
 running 334
 percent sign (%)
 beginning AREXX templates 261
 specifying binary numbers in assembler statement expressions 303
 period (.)
 grep wildcard character 225, 228, 230
 -pipe command-line option 97
 PLEN assembler directive 307
 plus sign (+)
 grep search pattern 226, 231
 pointers, dumping 63–64
 portname option, scmsg Project menu 258
 pound sign (#)
 ANSI # and ## operators not supported by define command 141
 indicating comments in smakefiles 274
 inserting before undefine command 191
 prev command, SC_SCMSG AREXX port 263
 Pri field, tasks command 109
 printing expressions
 See grep utility
 proceed command 46–47, 168
 See also ps command
 assigned to Return key 29
 compared with trace command 44, 46–47
 definition 29, 39
 examples 47, 168
 integer parameter 47
 syntax 46, 168

profile scripts
 See also cprinit file
 suppressing execution with -noprofile
 option 79–80

program counter-relative addressing
 examples 314
 forcing 311

program execution, controlling
 See also debugging programs
 break command 48–53
 clearing watches and watch breaks 34
 continuing execution 30
 executing until variable changes value
 33–34
 executing up to breakpoints 32
 go command 26, 30, 32, 39–44
 proceed commands 29, 46–47
 setting breakpoints 32, 47–53
 single stepping 29, 44–47
 stepping into called functions 29
 stepping over code 29–30
 trace commands 45–46
 using watch and watch break commands
 32–33

Project menu, scmsg utility 257–259

Prototype Generation Options window 267

ps command 169
 assembly mode 47
 example 169
 syntax 46, 169

public screen, specifying 80

public symbol 250

pull-down menus 20–22
 See also specific menus
 list of pull-down menus and command-line
 equivalents 20–22

Q

-q option
 diff utility 220
 grep utility 225
 smake utility 290

quiet option, break command 52–53, 135

quit command 170
 quitting the debugger 10, 34
 SC_SCMSG AREXX port 263
 syntax and description 170
 terminating CPRX 100

quit option, scmsg utility 255

quotes, double (“ ”)
 defining aliases 90
 hiding macros 92
 surrounding character strings 18
 using in grep patterns 227

quotes, single (‘ ’)
 enclosing literals in assembler statement
 expressions 303
 surrounding character strings 19

R

r command, oml utility 252

-r option
 tb utility 295

radix option, opt command 86, 167

range parameter 121–122

rangelen option, opt command 87, 167

ranges
 caution when using variables for setting
 watches 71
 dumping 62–63

re-attaching tasks 112

redirecting output (>)
 diff utility 219
 grep utility 224
 lprof utility 242
 lstat utility 244
 omd utility 249
 oml utility 251
 splat utility 291
 tb utility 294

REG assembler directive 307

register command 66–68, 171
 See also fregister command
 altering register contents 67–68
 determining variable values 77
 displaying register contents 67
 examples 171
 syntax 66–67, 171

register-direct addressing modes 313

register parameter 122–123

Register window 15, 81

registers
 altering contents 67–68
 displaying contents 67
 effect of opt task command 110
 modifying 110

- registers (*continued*)
 - optimizing 329–330
 - saving contents before retrieving
 - arguments from stack 317
 - scratch registers 318
- relocatable expressions 304
- RemTask function 114
- resident libraries, debugging 105–106
 - cautions for setting breakpoints 106, 110
 - example 106
 - libraries.cpr AREXX macro 106
 - limitations 106
 - setting breakpoints 105
 - symload command 106
- reslib option, opt command 88–89, 167
- restart command 172–173
 - compared with start command 181
 - crashes caused by improper use 26
 - definition 26, 36
 - examples 173
 - running go command before using 30
 - syntax and description 172
- restoring previously saved options in scopts utility 269
- results variable, AREXX 103
- return codes available to AREXX macros 103
- return command 174
- Return key for executing proceed
 - command 29
- REXX language
 - See AREXX interface
- rexonly command, SC_SCMSG AREXX port 264
- rexonly option, scmsg utility 256
- rflag command 175
- right-Amiga key 23
- RORG assembler directive 307
- run environment 6
- Run menu 21

S

- s option
 - asm command 326
 - grep utility 225
 - oml utility 253
 - smake utility 290
- splat utility 291
- tb utility 295
- sample CodeProbe session
 - See CodeProbe sample session
- sample programs
 - commands for running sample programs 36
 - lines.c 25
 - sample program 1 37–39
 - sample program 2 53–56
 - sc.examples drawer 25
- saving option settings in scopts utility 268–269
- sc command
 - adding addsym option to slink
 - automatically 8
 - debug=symbolflush option 9, 25
 - link option 9
 - parameters=register option 319
 - typical usage 9
- SC_CPR 101
 - See also AREXX macros
- SC_SCMSG AREXX port 262–264
- scmsg utility 255–265
 - autoedit option 255
 - Build menu 259
 - command options 255–256
 - config= option 255
 - Edit menu 260
 - format of messages 256–257
 - hidden option 255
 - Project menu 257–259
 - quit option 255
 - rexonly option 256
 - syntax and description 255–257
- scopts utility 266–270
 - entering options on command line 269
 - exiting without saving changes 269
 - gadgets 267–268
 - restoring previously saved options 269
 - saving option settings 268–269
 - tasks performed by 269–270
 - windows for setting options 266–267
- scratch registers 318
- screen command-line option 80
- script files
 - See cprinit file
 - See profile scripts
- scsetup utility 9, 271–272
- search command 176

- search option, opt command 87–88, 166
- search path 87–88
- searching for expressions
 - See grep utility
- searching for patterns
 - See splat utility
- SECTION assembler directive 309–310
 - definition 307
 - format 309
 - specifying addresses 309–310
- segment lists
 - assigning pointers 113
 - determining address 112
- select command, SC_SCMSG AREXX port 264
- semicolon (;)
 - escaping with backslash (\) in echo command 149
 - expanding alias names 90
 - preceding with escape character 93
 - using in cmd_list parameters 50
- serial ports
 - using for cross-debugging 97, 98
- SET assembler directive 307
- set command 177–178
- .SET fake target 282–283
- setenv command 11
- SetFunction 114
- setting up new projects
 - See scsetup utility
- Shell
 - starting across serial line 11
- shortcut keys 23–24, 29, 34
- shortcuts for command names 16
- show [activate] command, SC_SCMSG AREXX port 264
- SigWait field, tasks command 109
- .SILENT fake target 283
- single-stepping through programs 29, 44–47
- size field, assembly language source line 300
- size suffixes, assembler statements 301–302
- slash (/)
 - second character in Dialog window's title bar 17
- sleep command 179
- slink
 - adding or stripping debugging information 7
- addsym option 8, 9
- generating H_SYMBOL hunk 8
- stripdebug option 77
- smake utility 273–290
 - a option 289
 - actions performed by 274
 - alternate targets 283–284
 - b option 289
 - c option 289
 - comments in smakefiles 274
 - creating and using default files 288
 - creating smakefiles 274–276
 - d option 289
 - default (.def) files 287–288
 - default macros 279–280
 - defining transformation rules 280–281
 - e option 289
 - f option 289
 - fake targets 281–283
 - file dependencies, example 275
 - h option 290
 - i option 290
 - k option 290
 - local input files 284–286
 - macros 277–281
 - multiple targets 283
 - n option 290
 - options 289–290
 - overriding macro definitions 278
 - p option 290
 - q option 290
 - running smake 288–290
 - s option 290
 - smake.def file 287–288
 - special symbols 277
 - syntax 273, 289
 - t option 290
 - terminology 273
 - u option 290
 - x option 290
- smake.def file 287–288
- smakefile 273
- source command 78, 180
- source file
 - specifying location 77–78
- source line fields, assembly language 300–301
 - comment 301
 - label 300
 - operand 300

- source line fields, assembly language
(*continued*)
 - operation 300
 - size 300
 - source modes 83–84
 - assembly mode 45, 46, 47, 83
 - C mode 83
 - definition 27
 - illustrations 26
 - mixed mode 46, 83–84, 336
 - setting 27–28, 83–84
 - syntax of opt source command 83
 - source option, opt command 83–84, 166
 - Source window
 - functions 15
 - illustrations 26
 - open by default 12, 26
 - specifying start-up window coordinates 81
 - sourceis option 77–78
 - SPC assembler directive 307
 - special keys 23
 - speed command-line option 97
 - splat utility 291–293
 - d option 291
 - description 291
 - examples 291–293
 - o option 291
 - s option 291
 - syntax 291
 - v option 291
 - stack, assembly language programs 317
 - stack variables 110
 - StackPtr field, tasks command 109
 - start command 181–182
 - See also restart command
 - starting the debugger 25–34
 - startup command-line option 26, 80
 - State field, tasks command 109
 - state of the machine
 - See environment
 - statistics utilities
 - lprof utility 242–243
 - lstat utility 244–248
 - Step Into ResLib option
 - See reslib option, opt command
 - stepping through code
 - See program execution, controlling
 - strcat function 208
 - strcmp function 209
 - strcpy function 210
 - string parameter 123
 - display command 58
 - support for escape characters 123
 - strings, dumping 65–66
 - strings gadget 267
 - stripdebug option 77
 - strlen function 211
 - strlen option, opt command 86, 167
 - structures
 - displaying 30–31, 57, 59–60
 - setting watches or watch breaks 71
 - subrange parameter 119, 123–124
 - symbol 7
 - See also debug=symbol option
 - global symbol information 8
 - symbol command 183
 - symbol= option, gst utility 238
 - symbol tables, global
 - See gst utility
 - See hypergst utility
 - symfile option 98
 - symload command 112–113, 184
 - debugging resident libraries 106
 - definition 107
 - examples 184
 - syntax and description 112–113, 184
- ## T
- t option
 - oml utility 253
 - smake utility 290
 - t= option, lstat utility 244, 245–248
 - target files, smake utility 273
 - alternate targets 283–284
 - fake targets 281–283
 - multiple targets 283
 - target machine 7, 96
 - files required 96
 - task addresses 111
 - task-ID 110
 - task option, opt command 107, 110–111, 167
 - tasks
 - See also multitasking programs,
 - debugging
 - displaying current task 89
 - turning task catching on or off 89

- tasks command 109–110, 185
 - Address field 109
 - all option 110, 112
 - Debug field 109
 - definition 107
 - displaying tasks, example 109
 - examples 185
 - finding specific tasks 112
 - Name field 109
 - Pri field 109
 - SigWait field 109
 - StackPtr field 109
 - State field 109
 - syntax and description 185
 - Type field 109
- tb utility 294–296
 - description 294–295
 - examples 295–296
 - l option 295
 - m option 295
 - r option 295
 - s option 295
 - syntax 294
 - u option 295
 - v option 295
 - x option 295
- temp option, break command 52–53, 135
- terminating programs and tasks
 - See also quit command
 - CodeProbe tasks 108
 - CPRK and CPRX 99
 - oml utility 251
 - scmsg utility 255
- text command, SC_SCMSG AREXX port 264
- tool types, adding 78
- top command, SC_SCMSG AREXX port 264
- trace command 45–46, 186, 187
 - compared with proceed command 44, 46–47
 - debugging programs 45–46
 - definition 39
 - examples 45–46, 186
 - integer parameter 45
 - stepping into a called function 29
 - syntax and description 45, 186
- trace option, break command 51, 52–53, 135
- traceback information, displaying
 - See tb utility

- transformation rules for macros, smake
 - utility 280–281
- trap handling 113–114
- ts command 45, 187
- TTL assembler directive 308
- Type field, tasks command 109
- type parameter 124

U

- u option
 - asm command 326
 - smake utility 290
 - tb utility 295
- unalias command 92, 188
- unassemble command 189–190
- unassemble option, opt command 86, 167
- undefine command 94, 191
- underscore (_)
 - replaced by @ sign for compiler 318
- unit command-line option 97
- unload option, gst utility 238
- user environment 6
- utilities
 - See specific utilities

V

- v option
 - grep utility 225
 - oml utility 253
 - tb utility 295
- V option, grep utility 225
- variable parameter 125
- variables
 - assembler statement expressions 302
 - displaying value and type 32
 - dumping 61–62, 75
- verbose option, gst utility 238
- View menu 21

W

- w command-line option 80
- w option
 - asm command 326
 - diff utility 220

- wait option, scmsg Project menu 258
- watch breaks 32–34
 - clearing 34
 - creating 33
 - example 34
 - function 70
 - marked by exclamation mark (!) 33
 - setting on structures or arrays 71
 - setting ranges 71
 - slow execution while using 33, 72
 - static by default 70
- watch command 70–72, 192
 - See also wbreak command
 - See also wclear command
 - See also wdisable command
 - See also wenable command
 - See also wlist command
 - creating watches 33
 - definition 69
 - examples 192
 - syntax and description 70, 192
- watch command-line option 81
- Watch menu 22
- Watch window
 - displaying watches and watch breaks 69, 70
 - functions 15
 - opening 33
 - specifying start-up window coordinates 81
- watches 26, 69
 - caution when using variables for ranges 71
 - clearing 34
 - creating 33
 - dynamic by default 70
 - function 69–70
 - setting on structures or arrays 71
- wb command-line option 11, 80
- wbreak command 70–72, 193
- wclear command 34, 194
- wdialog command-line option 81
- wdisable command 195
- wenable command 196
- whatis command 68–69, 197
 - displaying value and type of variables 32
 - examples 68–69, 197
 - syntax and description 68, 197
- when option
 - break command 49, 135
 - go command 40, 42–44, 156
 - where command 110, 198
- window command 199
- windowing interface 12–16
 - See also specific windows
 - functions of CodeProbe windows 14–16
 - opening windows 15
 - screen layout 12–13
- windows displayed in scopts utility 266–267
- wlist command 69, 71, 200
- wmsg command 201
- Workbench process, invoking 80
- Workbench screen
 - invoking CodeProbe 11
 - running CodeProbe 10–11
 - specifying with -screen command-line option 80
 - specifying with -w command-line option 80
- wregister command-line option 81
- wsource command-line option 81
- wwatch command-line option 81

X

- x command, oml utility 252
- x option
 - omd utility 249
 - oml utility 253
 - smake utility 290
 - stripping debug information 96
 - tb utility 295
 - using with CPRX 99
- XDEF assembler directive 308
 - defining symbols in data section 309
 - referencing global data 322–323
- XREF assembler directive 308
 - defining symbols in data section 309
 - referencing global data 322–323
 - using with CSECT directive 311–312

Z

- z option, lstat utility 244

Special Characters

{} (braces)
 See braces ({})
 \ (backslash character)
 See backslash character (\)
 / (slash)
 See slash (/)
 = (equals sign)
 See equals sign (=)
 [] (brackets)
 See brackets ([])
 . (period)
 See period (.)
 < (less than sign)
 See less than sign (<)
 + (plus sign)
 See plus sign (+)
 & (ampersand operator)
 See ampersand operator (&)
 & (ampersand symbol)
 See ampersand symbol (&)
 & (and operator)
 See and operator (&)
 ! (exclamation mark)
 See exclamation mark (!)
 \$ (dollar sign)
 See dollar parameter (\$)
 See dollar sign (\$)
 * (asterisk)
 See asterisk (*)
 ; (semicolon)
 See semicolon (;)
 ^ (caret)
 See caret (^)
 - (hyphen)
 See hyphen (-)
 - (minus sign)
 See minus sign (-)
 , (comma)
 See comma (,)
 % (percent sign)
 See percent sign (%)
 _ (underscore)
 See underscore (_)
 > (greater than sign)
 See greater than sign (>)
 ` (back-tick character)
 See back-tick character (`)
 # (pound sign)
 See pound sign (#)

@ (at sign)
 See at sign (@)
 ' (single quotes)
 See quotes, single (' ')
 " (quotes)
 See quotes, double (" ")

Your Turn

If you have comments or suggestions about the *SAS/C Development System User's Guide, Volume 2: Debugger, Utilities, Assembler* or the SAS/C Development System, please send them to us on a photocopy of this page.

Please return the photocopy to the Publications Division (for comments about this book) or the Technical Support Division (for suggestions about the SAS/C Development System) at SAS Institute Inc., SAS Campus Drive, Cary, NC 27513.



**This was brought to you
from the archives of**

<http://retro-commodore.eu>